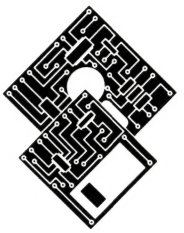
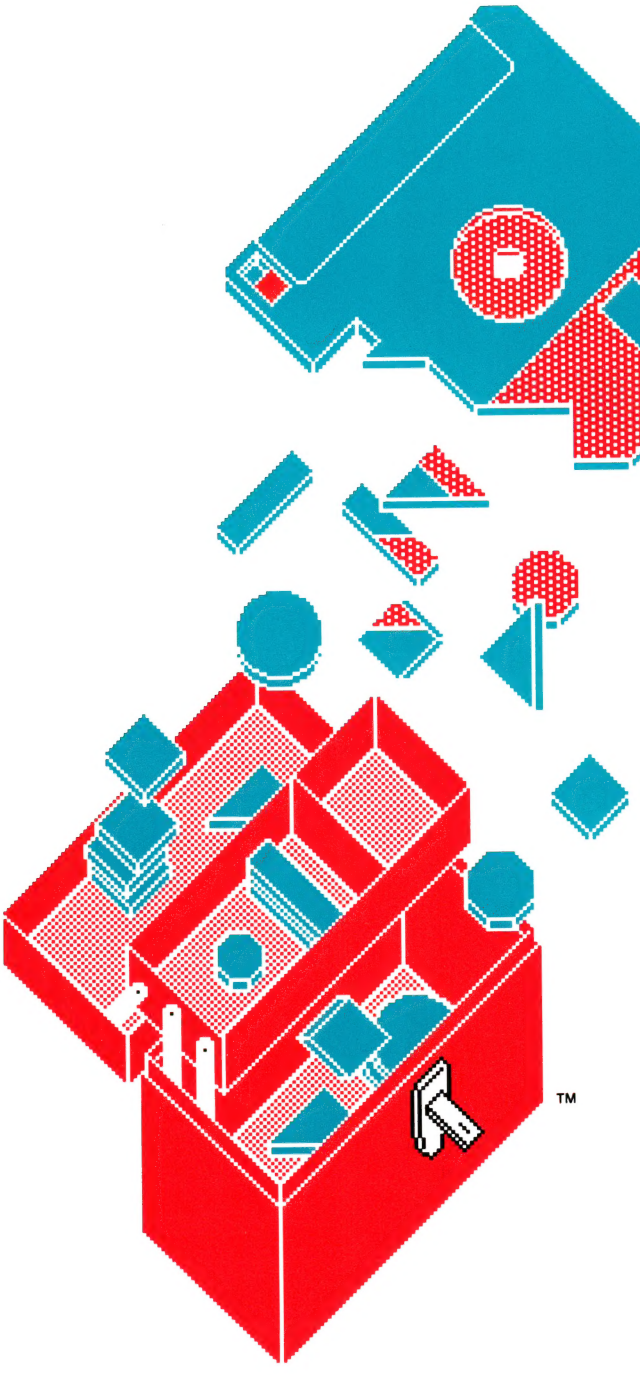


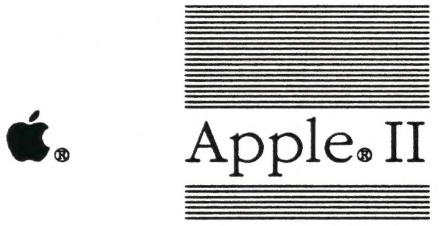
APPLE
PROGRAMMER'S
AND DEVELOPER'S
ASSOCIATION



Apple IIGS Programmer's Workshop C

Version 1.0

APDA# K2B001



Apple IIGS Programmer's Workshop C Reference

APPLE COMPUTER, INC.

This manual and the software described in it are copyrighted, with all rights reserved. Under the copyright laws, this manual or the software may not be copied, in whole or part, without written consent of Apple, except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format.

You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

© Apple Computer, Inc., 1985-88
20525 Mariani Ave.
Cupertino, California 95014
(408) 996-1010

© AT&T, 1985

Apple, the Apple logo, Apple IIGS, LaserWriter, Macintosh, and ProDOS are registered trademarks of Apple Computer, Inc.

SANE is a trademark of Apple Computer, Inc.

UNIX is a registered trademark of AT&T.

DEC, VAX, and PDP are trademarks of Digital Equipment Corporation.

IBM is a registered trademark of International Business Machines Company.

NS16000 is a trademark of National Semiconductor Corporation.

Z8000 and Z8070 are trademarks of Zilog Corporation.

Simultaneously published in the United States and Canada.

Contents

Figures and tables viii

Preface **About this manual ix**

A road map to the Apple IIGS technical manuals ix
- Introductory manuals xi
  - The technical introduction xi
  - The programmer's introduction xi
- Machine reference manuals xi
  - The hardware reference manual xii
  - The firmware reference manual xii
- The toolbox manuals xii
- The Programmer's Workshop manual xii
- Programming-language manuals xiii
- Operating-system manuals xiii
- All-Apple manuals xiii

How to use this manual xiv
- What this manual contains xiv

Visual cues xv
- New terms xv
- Notes and warnings xv

Language notation xv

Other reference materials you'll need xvi

Part I: Programmer's guide

Chapter 1: Overview 1-1

About the Apple IIGS Programmer's Workshop 1-2
- The APW Shell 1-2
- The APW Editor 1-3
- The APW Linker 1-3

About APW C 1-3
- Mode of operation 1-4
- Standard Apple Numeric Environment 1-4
- Object module format 1-4

About the Apple IIGS system software 1-5

What you need 1-5

APW C concepts 1-6
- Relocatable load files 1-6
- Program segmentation 1-8
- Dynamic segments 1-12

Library files 1-13
Program interactions 1-14
Using the APW C libraries 1-17

Chapter 2: Using the APW C Compiler 2-1

Installing APW C 2-2
Backing up your APW C disk 2-2
Installation 2-2
Running APW C on 3.5-inch disks 2-3
Writing and running a sample program 2-4
Writing the sample program 2-4
Compiling and linking the sample program 2-5
Running the sample program 2-5
The APW C Compiler 2-5
The compilation process 2-5
Suspending or canceling the compilation 2-6
C compiler error messages 2-6
C compiler shell commands 2-6
Editing a source file 2-6
Compiling and linking a program 2-7
Command notation 2-7
CC 2-9
CHANGE 2-9
CMPL 2-9
CMPLG 2-9
COMPILE 2-10
EDIT 2-13
LINK 2-14
RUN 2-15
Examples of these commands 2-16
Appending files 2-16
Partial compilation or assembly 2-17
The linker 2-17
Making a library 2-17
Files for compiling and linking 2-17
Include-file search rules 2-18
Library files 2-18

Chapter 3: Sample Programs 3-1

General procedure 3-2
Writing and editing the sample source code 3-3
Creating object code: compiling and assembling 3-5
Creating load files: linking 3-6
Running your program 3-7
Creating a compact load file 3-7
Building a larger application: BONES 3-8
Writing desk accessories in APW C 3-8
Writing new desk accessories in APW C 3-8
A sample C desk accessory 3-10

Part II: Language Reference
Chapter 4: The APW C Language 4-1
Language definition 4-1
Variable names 4-1
Data types 4-1
Numeric constants 4-3
Type `void` 4-3
Type `enum` 4-4
Register variables 4-5
Structures 4-5
Reserved symbols 4-6
Standard Apple Numeric Environment extensions 4-6
Constants 4-7
Expressions 4-7
Comparison involving a NaN 4-8
Parameters and function results 4-8
Numeric input and output 4-8
Numeric environment 4-8
About the SANE routines in CLIB 4-8
Programming with IEEE arithmetic 4-9
The in-line assembler 4-9
In-line assembly-code declarations and definitions 4-10
In-line assembler syntax 4-10
Pascal-style functions 4-12
Pascal-style function declarations 4-13
The `inline` declaration 4-13
Pascal-style function definitions 4-14
Pascal-style strings: `\p` 4-14
Parameter and result data types 4-15
Global and external data types 4-16
How parameters are passed 4-16
C-style functions 4-16
Pascal-style functions 4-17
Sample program 4-17
Implementation notes 4-19
Size and byte-alignment of variables 4-19
Byte ordering 4-19
Variable allocation 4-19
Variables of type `void` 4-19
Array indexing 4-19
Types `unsigned char`, `unsigned short`, and `unsigned long` 4-21
Bit fields 4-21
Evaluation order 4-21
String substitutions in `define` statements 4-21
Assignment operators 4-22
Language anachronisms 4-22
Assignment operators 4-22
Initialization 4-22
Compiler limitations 4-22
Performance tips 4-23
The `segment` command 4-23
The `#append` directive 4-23
`START.ROOT`, restartability, and `StandAlone` 4-23
Code-generation memory model 4-24

Chapter 5: The Standard C Library 5-1

About the Standard C Library 5-2
Error numbers 5-3
abs—return integer absolute value 5-5
atof—convert ASCII string to floating-point number 5-6
atoi—convert string to integer 5-7
close—close a file descriptor 5-8
conv—translate characters 5-9
creat—create a new file or rewrite an existing file 5-10
ctype—classify characters 5-11
dup—duplicate an open file descriptor 5-13
ecvt—convert a floating-point number to a string 5-14
exit—terminate the current application 5-15
exp—exponential, logarithm, power, square-root functions 5-16
faccess—named file access and control 5-17
fclose—close or flush a stream 5-18
fcntl—file control 5-19
ferror—ferror status inquiries 5-20
floor—floor, ceiling, mod, absolute value functions 5-21
fopen—open a buffered file stream 5-22
fread—binary input/output 5-24
frexp—manipulate parts of floating-point numbers 5-25
fseek—reposition a file pointer in a stream 5-26
getc—get a character or a word from a stream 5-27
getenv—access exported APW Shell variables 5-28
gets—get a string from a stream 5-29
hypot—Euclidean distance function 5-30
ioctl—control a device 5-31
lseek—move read/write file pointer 5-33
malloc—memory allocator 5-34
memory—memory operations 5-36
onexit—install a function to be executed at program termination 5-37
open—open for reading or writing 5-38
printf—print formatted output 5-39
putc—put character or word on a stream 5-42
puts—write a string to a stream 5-43
qsort—quicker sort 5-44
rand—a simple random-number generator 5-45
read—read from file 5-46
scanf—convert formatted input 5-47
setbuf—assign buffering to a stream 5-51
setjmp—nonlocal transfer of control 5-53
sinh—hyperbolic functions 5-54
stdio—standard buffered input/output package 5-55
string—string operations 5-58
strtol—convert a string to a long 5-60
trig—trigonometric functions 5-61
ungetc—push a character back into the input stream 5-62
unlink—delete a named file 5-63
write—write on a file 5-64

Chapter 6: Shell Calls 6-1

How to make a shell call 6-2
How a program makes a shell call 6-3
Call descriptions 6-3
GET_LINFO and SET_LINFO 6-3
GET_LANG 6-6
SET_LANG 6-6
ERROR 6-7
SET_VAR 6-7
VERSION 6-7
READ_INDEXED 6-8
INIT_WILDCARD 6-8
NEXT_WILDCARD 6-9
GET_VAR 6-9
EXECUTE 6-10
DIRECTION 6-11
REDIRECT 6-11
STOP 6-12
WRITE_CONSOLE 6-12

Appendix A: Calling Conventions A-1

C calling conventions A-1
Parameters A-1
Function results A-1
Register conventions A-2
Pascal-style calling conventions A-2
Parameters A-2
Function results A-2
Register conventions A-2

Appendix B: Files supplied with APW C B-1

Appendix C: Comparison with Macintosh Programmer'sWorkshop C C-1

Data types C-1
Register variables C-1
Structured variables C-1
Pascal-compatible function declarations C-2
Preprocessor statements C-2
Dangling case in switch statements C-3
In-line assembly-code declarations C-3

Appendix D: Library Index C-1

Appendix E: ASCII Table E-1

Appendix F: APW C Compiler Error Messages F-1

Glossary GL-1
Index IN-1

Figures and tables

Preface **About this manual ix**

Figure P-1 A roadmap to the technical manual x
Table P-1 The Apple IIGS technical manuals ix

Chapter 1 **Overview 1-1**

Figure 1-1 Creating an executable C program on the Apple IIGS 1-8
Figure 1-2 Creating object segments in source code 1-9
Figure 1-3 Assigning load segments in source code 1-10
Figure 1-4 Relationship between objects segments and load segments 1-11
Figure 1-5 Relationship between object files and library files 1-14
Figure 1-6 Program interactions 1-15
Figure 1-7 APW C library interactions 1-17

Chapter 2 **Using the APW C Compiler 2-1**

Table 2-1 Include-file search rules 2-18

Chapter 3 **Sample Programs 3-1**

Table 3-1 Tool sets loaded and available to new desk accessories 3-9

Chapter 4 **The APW C Language 4-1**

Table 4-1 Size and range of data types 4-2
Table 4-2 Parameter and result data types 4-15

Chapter 6 **Shell calls 6-1**

Table 6-1 Shell calls

Preface

About This Manual

This manual contains the information about Apple® IIGS® Programmer's Workshop C that you need when writing C programs for the Apple IIGS computer. It assumes that most readers already know the C programming language, as defined in Kernighan and Ritchie's *The C Programming Language*. For this reason, it does not repeat their definition of the C language, but instead defines the differences between APW C and "K and R" C. However, this manual can also be used by those learning C for the first time. The introductory chapters tell how to write, compile, link, and run a simple C program. From there, you can follow Kernighan and Ritchie's book or any other standard textbook on C.

A road map to the Apple IIGS technical manuals

The Apple IIGS personal computer has many advanced features, making it more complex than earlier models of the Apple II computer. To describe the Apple IIGS fully, Apple has produced a suite of technical manuals. Depending on the way you intend to use the Apple IIGS, you may need to refer to a select few of the manuals, or you may need to refer to most of them.

The technical manuals are listed in Table P-1. Figure P-1 is a diagram showing the relationships between the different manuals.

Table P-1
The Apple IIGS technical manuals

Title	Subject
Technical Introduction to the Apple IIGS	What the Apple IIGS is
Apple IIGS Hardware Reference	Machine internals—hardware
Apple IIGS Firmware Reference	Machine internals—firmware
Programmer's Introduction to the Apple IIGS	Concepts and a sample program
Apple IIGS Toolbox Reference, Volume 1	How the tools work and some toolbox specifications
Apple IIGS Toolbox Reference, Volume 2	More toolbox specifications
Apple IIGS Programmer's Workshop Reference	The development environment
Apple IIGS Programmer's Workshop Assembler Reference	Using the APW Assembler
Apple IIGS Programmer's Workshop C Reference	Using C on the Apple IIGS
ProDOS 8 Technical Reference Manual	ProDOS for Apple II programs
Apple IIGS ProDOS 16 Reference	ProDOS and loader for Apple IIGS
Human Interface Guidelines: The Apple Desktop Interface	Guidelines for the desktop interface
Apple Numerics Manual	Numerics for all Apple computers

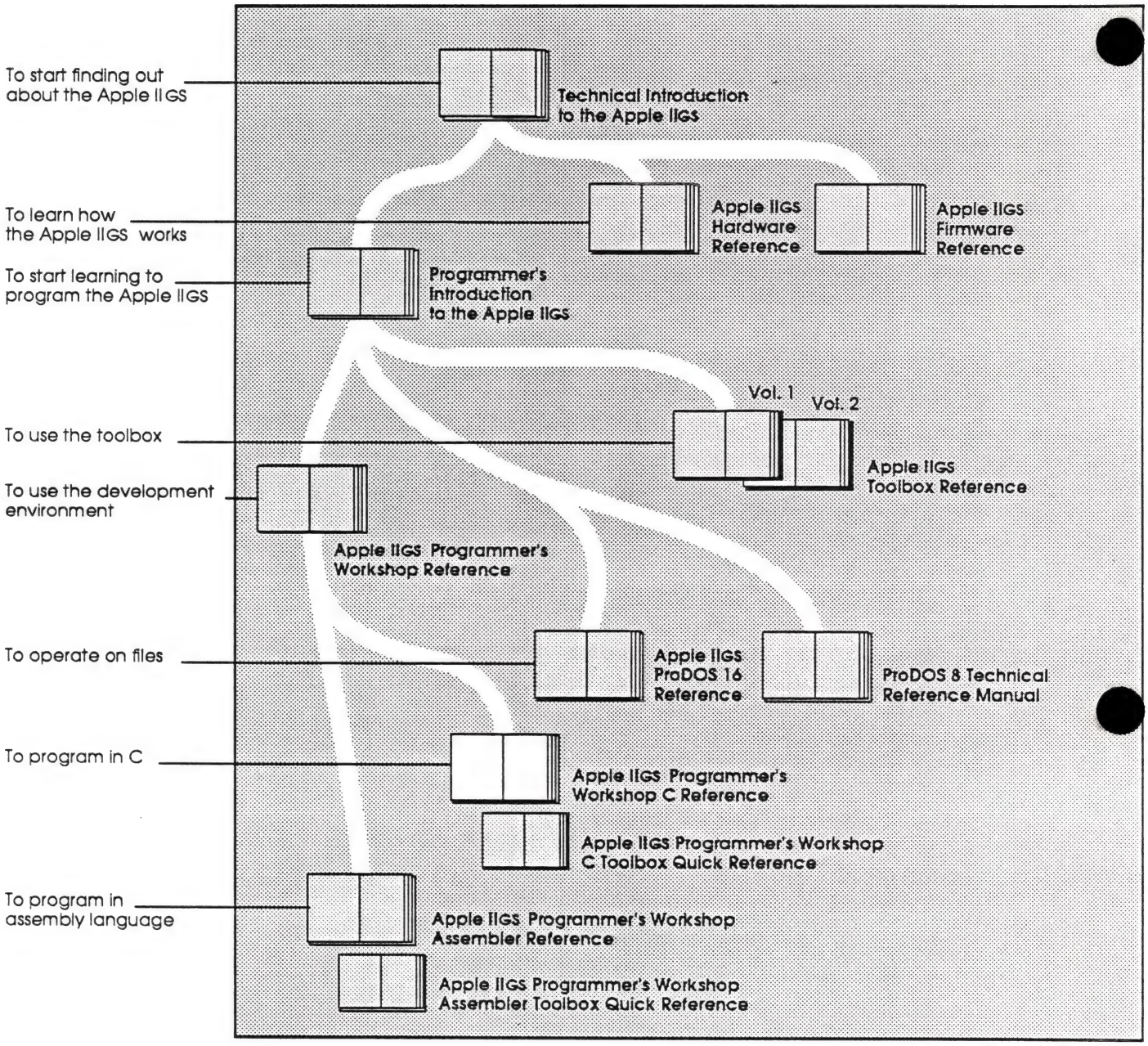

To start finding out about the Apple IIGS
Technical Introduction to the Apple IIGS
To learn how the Apple IIGS works
Apple IIGS Hardware Reference
Apple IIGS Firmware Reference
To start learning to program the Apple IIGS
Programmer's Introduction to the Apple IIGS
Vol. 1
Vol. 2
To use the toolbox
Apple IIGS Toolbox Reference
To use the development environment
Apple IIGS Programmer's Workshop Reference
To operate on files
Apple IIGS ProDOS 16 Reference
ProDOS 8 Technical Reference Manual
To program in C
Apple IIGS Programmer's Workshop C Reference
Apple IIGS Programmer's Workshop C Toolbox Quick Reference
To program in assembly language
Apple IIGS Programmer's Workshop Assembler Reference
Apple IIGS Programmer's Workshop Assembler Toolbox Quick Reference

Figure P-1
A road map to the technical manuals

The following sections briefly describe the manuals listed in Table Pref-1 and Figure Pref-1.

Introductory manuals

These books are introductory manuals for developers, computer enthusiasts, and other Apple IIGS owners who need technical information. As introductory manuals, their purpose is to help the technical reader understand the features of the Apple IIGS, particularly the features that are different from other Apple computers. Having read the introductory manuals, the reader will refer to specific reference manuals for details about a particular aspect of the Apple IIGS.

The technical introduction

The *Technical Introduction to the Apple IIGS* is the first book in the suite of technical manuals about the Apple IIGS. It describes all aspects of the Apple IIGS, including its features and general design, the program environments, the toolbox, and the development environment.

Where the *Apple IIGS Owner's Guide* is an introduction from the point of view of the user, the *Technical Introduction to the Apple IIGS* describes the Apple IIGS from the point of view of the program. In other words, the manual describes the things the programmer has to consider while designing a program, such as the operating features the program uses and the environment in which the program runs.

You should read the *Technical Introduction to the Apple IIGS* no matter what kind of programming you intend to do, because it will help you understand the powers and limitations of the machine. If you are going to be doing assembly-language or system programming, this book is essential. To find out all about any one aspect of the Apple IIGS, you should read one of the following specific technical manuals.

The programmer's introduction

When you start writing programs that use the Apple IIGS user interface (with windows, menus, and the mouse), the *Programmer's Introduction to the Apple IIGS* provides the concepts and guidelines you need. It is not a complete course in programming, but is only a starting point for programmers writing applications for the Apple IIGS. It introduces the routines in the Apple IIGS Toolbox and the program environment they run under. The manual includes a sample **event-driven program** that demonstrates how a program uses the toolbox and the operating system.

Machine reference manuals

There are two reference manuals for the machine itself: the *Apple IIGS Hardware Reference* and the *Apple IIGS Firmware Reference*. These books contain detailed specifications for people who want to know exactly what's inside the machine.

If you are doing system programming or are writing programs that are designed to recognize whether they are running on the Apple IIGS or older Apple II computers, these books are essential.

The hardware reference manual

The *Apple IIGS Hardware Reference* is required reading for hardware developers, and is also of interest to anyone else who wants to know how the machine works. Information for developers includes the mechanical and electrical specifications of all connectors, both internal and external. Information of general interest includes descriptions of the internal hardware, which provide a better understanding of the machine's features.

The firmware reference manual

The *Apple IIGS Firmware Reference* describes the programs and subroutines that are stored in the machine's read-only memory (ROM), with two significant exceptions: Applesoft BASIC and the toolbox, which have their own manuals. The *Apple IIGS Firmware Reference* includes information about interrupt routines and low-level I/O subroutines for the serial ports, the disk port, and for the Desktop Bus interface, which controls the keyboard and the mouse. The reference also describes the Monitor, a low-level programming and debugging aid for assembly-language programs.

The toolbox manuals

Like the Macintosh, the Apple IIGS has a built-in toolbox. The *Apple IIGS Toolbox Reference,* Volume I, introduces concepts and terminology, and tells how to use some of the tools. The *Apple IIGS Toolbox Reference,* Volume II, contains information about the rest of the tools, and describes how to write and install your own tool set.

Of course, you don't have to use the toolbox at all. If you only want to write simple programs that don't use the mouse, windows, menus, or other parts of the **desktop user interface,** then you can get along without the toolbox. However, if you are developing an application that uses the desktop interface, or if you want to use the Super Hi-Res graphics display, you'll find the toolbox indispensable.

The Programmer's Workshop manual

The development environment on the Apple IIGS is the Apple IIGS Programmer's Workshop (APW). APW is a set of programs that enable developers to create and to debug application programs on the Apple IIGS. The *Apple IIGS Programmer's Workshop Reference* includes information about the parts of the workshop that all developers will employ, regardless which programming language they use: the shell, the editor, the linker, the debugger, and the utilities. The manual also tells how to write other programs, such as custom utilities and compilers, to run under the APW Shell. (For brevity, this text will usually refer to the *Apple IIGS Programmers Reference* as the *APW Reference.*)

The *APW Reference* describes the way you use the workshop to create an application and includes a sample program to show how this is done.

Programming-language manuals

Apple currently provides a 65816 assembler and a C compiler. Other compilers can be used with the workshop, provided that they follow the standards defined in the *APW Reference*.

There is a separate reference manual for each programming language on the Apple IIGS. Each manual includes the specifications of the language and of the Apple IIGS libraries for the language, and describes how to write a program in that language. The manuals for the languages Apple provides are the *Apple IIGS Programmer's Workshop Assembler Reference* and the *Apple IIGS Programmer's Workshop C Reference.*

Operating-system manuals

There are two operating systems that run on the Apple IIGS: ProDOS 16 and ProDOS 8. Each operating system is described in its own manual: *ProDOS 16 Technical Reference Manual* and *Apple IIGS ProDOS 8 Reference.* ProDOS 16 uses the full power of the Apple IIGS and is not compatible with earlier Apple II computers. The ProDOS 16 manual includes information about the System Loader, which works closely with ProDOS 16. If you are writing programs for the Apple IIGS, whether as an application programmer or as a system programmer, you are almost certain to need the *ProDOS 16 Reference.*

ProDOS 8, previously just called *ProDOS,* is compatible with the models of Apple II that use 8-bit CPUs. As a developer of Apple IIGS programs, you need to use ProDOS 8 only if you are developing programs to run on 8-bit Apple II's, as well as on the Apple IIGS.

All-Apple manuals

In addition to the Apple IIGS manuals just mentioned, there are two manuals that apply to all Apple computers: *Human Interface Guidelines—The Apple Desktop Interface;* and the *Apple Numerics Manual.* If you develop programs for any Apple computer, you should know about these manuals.

The *Human Interface Guidelines* describes Apple's standards for the desktop interface to any program that runs on an Apple computer. If you are writing a commercial application for the Apple IIGS, you should be fully familiar with the contents of this manual.

The *Apple Numerics Manual* is the reference for the Standard Apple Numeric Environment (SANE™), a full implementation of the IEEE standard for floating-point arithmetic. The functions of the Apple IIGS SANE tool set match those of the Macintosh SANE packages and of the 6502 Assembly-Language SANE™ software. If your application requires accurate or robust arithmetic, you'll probably want to use the SANE routines in the Apple IIGS. The *Apple IIGS Toolbox Reference* tells how to use the SANE tool set routines in your programs. The *Apple Numerics Manual* is the comprehensive reference for the semantics of the SANE routines.

How to use this manual

If you are an experienced C programmer but have never written a program for the Apple IIGS, Chapters 1, 2, and 3 will give you enough information to get standard C programs running. (If you have written other programs for the Apple IIGS, Chapter 1 will be redundant.) The remaining chapters tell you what you need to write C programs that use the capabilities of the Apple IIGS.

If you are new to C, Chapter 1 will tell you what you need to go through a C textbook, such as Kernighan and Ritchie's, which you should read next. After you are familiar with C, you can learn about the capabilities of the C compiler and this particular implementation.

What this manual contains

This manual is divided into two major sections. Part I, "Programmer's Guide," introduces you to APW C and its programming environment.

- Chapter 1, "Overview," introduces the environment in which you'll use the C compiler. The chapter discusses the Apple IIGS Programmer's Workshop, ProDOS 16, and the Apple IIGS tools, and lists the hardware and software you'll need.
- Chapter 2, "Using the APW C Compiler," describes the compilation process, lists the shell commands you'll need working with the compiler, and discusses the linker, the debugger, and other utilities.
- Chapter 3, "Sample Program," takes you step-by-step through the process of building a C program that has an assembly-language subroutine.

Part II, "Language Reference," is a detailed description of the structure and components of the APW C and its libraries.

- Chapter 4, "The APW C Language," describes Apple extensions to C and clarifies aspects of the language definition as they apply to this implementation.
- Chapter 5, "The Standard C Library," documents functions for standard I/O, string manipulation, math routines, and other useful features not built into the language.
- Chapter 6, "Shell Calls," lists the C interfaces to the APW Shell.
- Appendix A, "Calling Conventions," tells how to write calls between C and Pascal.
- Appendix B, "Files Supplied with APW C," contains a list of all the files that are supplied with this product.
- Appendix C, "Comparison with Macintosh Workshop C," describes the differences between MPW C and APW C.
- Appendix D, "Library Index," is a combined index of identifiers in the Standard C Library and the Apple IIGS Interface Libraries.
- Appendix E, "ASCII Table," contains decimal, octal, and hexadecimal equivalents of each character in the Apple extended ASCII character set .

Visual cues

Certain conventions in this manual provide visual cues alerting you, for example, to the introduction of a new term and important or useful information. These are described in this section. Typographical conventions are described in the next section, "Language Notation."

New terms

When a new term is introduced, it is printed in **boldface** the first time it is used. Boldfacing lets you know that the term has not been defined earlier and that there is an entry for it in the glossary.

Notes and warnings

Special messages to note are marked as follows:

❖ *Note:* Text set off in this way presents sidelights or interesting points of information.

Important

Text set off in this way presents important information or instructions that you should read before proceeding.

Warning

A warning set off like this alerts you to something that could cause loss of data or damage to software.

Language notation

This manual uses certain conventions in common with other Apple IIGS language manuals.

□ Words and symbols that are part of the C language, as well as anything that you type on the keyboard or that can appear on the screen, are presented in a monospace font:

```
int ndigit[10]
```

□ Metalanguage expressions, which are used in syntax diagrams to indicate items that are replaced by C, are in italic:

```
else if (condition)
     statement
```

Here *condition* and *statement* are expressions that are replaced by actual C expressions`. The `else if` and the parentheses are C code.

In addition, the following conventions are observed:

Convention	Meaning
[ ]	Square brackets indicate that the enclosed item is optional.
. . .	A horizontal ellipsis indicates that the preceding item or items can be repeated as necessary.
. . .	A vertical ellipsis indicates that not all of the statements in an example or figure are shown.

Other reference material you'll need

In order to write C programs for the Apple IIGS, you'll need to be familiar with these additional reference materials:

- *Apple IIGS Programmer's Workshop Reference.* This book describes the APW environment in which the C compiler operates, including the shell, editor, linker, debugger, and other important utilities.
- *The C Programming Language,* by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, 1978). This is a standard reference book for the C language in its original form. Appendix A of this book is a formal definition of K and R C.
- *C: A Reference Manual,* by Samuel P. Harbison and Guy L. Steele (Prentice-Hall, 1985). This is a complete reference book for standard C, as implemented by the Portable C Compiler, including the Western Electric extensions to K and R C.
- *Apple IIGS Toolbox Reference, Volumes I and II.* These books contain everything you need to program using the Apple IIGS ROM and associated RAM routines. The two volumes cover windows, alert boxes, menus, graphics, the SANE tool set, and much more.
- *Apple Numerics Manual.* This book describes in detail the floating-point arithmetic used in Apple computers. See the *Toolbox Reference* for a detailed description of the calling sequence for SANE routines.

Part I

Programmer's Guide

Chapter 1

Overview

This chapter introduces the Apple IIGS Programmer's Workshop (APW). The first section, "About the Apple IIGS Programmer's Workshop," describes the various parts of APW. The second section, "About Apple IIGS System Software," describes ProDOS 16, the System Loader, and the Memory Manager. The third section, "What You Need," describes the hardware and software you need to run APW C. The fourth section, "APW C Concepts," describes the relationships between source, object, load, and library files. The fifth section, "Program Interactions," describes the process of building a program. The sixth section, "Using the APW C Libraries," shows the libraries that mediate between an application and the Apple IIGS.

About the Apple IIGS Programmer's Workshop

The Apple IIGS Programmer's Workshop is a suite of software designed to assist developers in writing Apple IIGS applications programs. This development environment includes a **command interpreter,** known as the **shell;** a text **editor;** a **linker;** and a set of utilities. APW supports C and 65816/65C02 assembly-language programming; other languages are planned. Further support for developers is provided by a comprehensive set of routines known as the **Apple IIGS Toolbox.** The toolbox routines are accessed from APW, but are not part of APW. For a comprehensive description of APW, refer to the *Apple IIGS Programmer's Workshop Reference*. For detailed information on the Apple IIGS Toolbox, refer to the *Apple IIGS Toolbox Reference: Volumes I and II.*

The APW Shell

The APW Shell provides the interface that allows you to work with the C compiler and perform tasks such as file, directory, and disk management. The shell also acts as an extension to ProDOS 16, providing several functions that can be called by programs running under the shell. The C compiler can use a set of shell calls to perform the following functions:

- pass parameters and operations flags between the shell and APW programs
- read the current language number
- set the current language number
- return the address of the command table
- get filenames using wildcards

APW C provides C interfaces to the shell calls. The calls and their C interfaces are discussed in Chapter 6, "Shell Calls."

Commands most often used while working with the C compiler are described in Chapter 2, "Using the APW C Compiler." The APW Shell is fully described in Chapters 2 and 3 of the *APW Reference*.

The APW Editor

The APW Editor is a full-screen text editor that operates under keyboard control.

You can send commands to the shell to perform tasks such as

- manipulating text
- searching for and replacing text strings
- moving your position in the file
- scrolling the screen
- setting and clearing tab stops
- defining and using keyboard macros

The APW Editor is fully described in Chapters 2 and 4 of the *APW Reference.*

The APW Linker

The APW Linker takes the **object files** produced by the C compiler and generates **load files** that the System Loader can load into memory. Although the linker is a single program, conceptually there are two APW linkers:

1. Normally the linker is called by a shell command, such as `LINK` or `CMPL` (compile or link). These commands provide a limited set of options, setting other options to default values. This linker is referred to as the **standard linker.**
2. Alternatively, all functions of the APW Linker can be controlled by compiling a file of linker commands. The linker command language, called *LinkEd,* allows you to do such things as place specific object-file segments in specific load-file segments, search specific libraries, and control linker printout. The aspect of the linker controlled by LinkEd files is called the **advanced linker.**

About APW C

APW C is a complete implementation of the C programming language. APW C consists of a C compiler, the Standard C Library, the Apple IIGS Interface Libraries, and the C SANE Library.

The C Programming Language by Kernighan and Ritchie is an authoritative written definition of C in its original form: this version of C is referred to as *K and R C.* However, the language has changed in several ways since the book was written. In addition, numerous details of the language definition are open to interpretation, with the result that the de facto standard definition of C differs in several ways from the language originally defined by Kernighan and Ritchie. This de facto standard is loosely defined by the most widely used implementation of C, the *Portable C Compiler* (PCC).

This manual, uses the term *Standard C* for C as defined and implemented by the Berkeley 4.2 BSD VAX implementation of PCC, including the documented Western Electric extensions: type `void`, enumeration data types, and structures as function parameters and results. *C: A Reference Manual,* by Harbison and Steele, describes Standard C fully. APW C is based on this de facto standard and not on the proposed ANSI standard currently under development.

Apple has extended Standard C to facilitate writing programs for the Apple IIGS. In addition to the Western Electric extensions, APW C includes a function modifier that allows calls to and from Pascal programs and the Apple IIGS Interface Libraries. APW C also supports the Standard Apple Numeric Environment (SANE), described later in this chapter.

Mode of operation

The APW C Compiler, and APW C itself, operates in the Apple IIGS's native mode. In **native mode,** the full instruction set of the 65816 processor is available to the compiler.

Standard Apple Numeric Environment

The APW C Compiler provides full support for the Standard Apple Numeric Environment (SANE). APW C and the SANE routines in CLIB compose a fully conforming implementation of extended-precision binary floating-point arithmetic as specified by IEEE Standard 754. This standard specifies data types, arithmetic, and conversions, as well as tools for handling exceptions such as overflow and division by zero. SANE supports all requirements of the IEEE standard and goes beyond the specifications of the standard by including a library of high-quality scientific and financial functions. Thus, SANE provides a numerics environment sufficient for a wide range of applications.

Source programs that use only the `float` and `double` types, and standard C operations compile and run without modification.

Object module format

The **object module format** (OMF) on the Apple IIGS is the general format used in object files, **library files,** and load files. On the Apple IIc and IIe, there is only one loadable file format, called the *binary file format,* which consists of one absolute memory image along with its destination address. On the Apple IIGS, object module format allows, while a program is running, dynamic loading and unloading of **load segments** containing program code and data. Additionally, each APW language produces its object code in the object module format, allowing you to link together subroutines written in different languages.

There are currently two OMFs: Version 1, produced by the APW Assembler, APW C Compiler, and APW Linker; and Version 2, produced when you run an executable load file through the Compact utility. To make an application written in C restartable, you must run Compact on the load file (or files) that contains the application.

About the Apple IIGS system software

System tasks are handled by ProDOS 16, the **System Loader,** and the **Memory Manager.** ProDOS 16 is the core, or kernel, of the Apple IIGS's operating system. It provides file management and input/output (I/O) capability.

Working closely with ProDOS 16, the System Loader is responsible for loading all code and data into the Apple IIGS memory. The System Loader is capable of static and dynamic loading and relocating of load segments.

The Memory Manager is responsible for allocating memory. It provides space for load segments, tells the System Loader where to place them, and moves segments within memory when additional space is needed.

ProDOS 16 and the System Loader are documented in the *Apple IIGS ProDos 16 Reference.* The Memory Manager is documented in both the *Apple IIGS ProDos 16 Reference* and the *Apple IIGS Toolbox Reference, Volumes I and II.*

What you need

To use the Apple IIGS Programmer's Workshop, you must have the following hardware and software. The Preface gives a list of Apple IIGS manuals that you will find useful.

- ☐ An Apple IIGS computer, or an Apple IIe computer with an installed Apple IIGS upgrade, with 256K bytes of RAM.
- ☐ An installed Apple IIGS memory-expansion card with 1 megabyte (1M-byte) of RAM. With this card, the Apple IIGS has 1280K of RAM.
- ☐ The 3.5-inch Apple IIGS System Disk.
- ☐ The two 3.5-inch APW disks.
- ☐ The 3.5-inch APW C disk that contains the files shown in Appendix B.
- ☐ Two 800K disk drives (only one is needed if you have a hard disk, but two are handy for such operations as copying disks).
- ☐ Disks containing any other APW languages you intend to use with this system. You must install the files on these disks onto the Apple IIGS disk as described in the manuals that came with them.

Important

APW requires 1M-byte of *available* memory. That means that if you have 1280K of RAM in your Apple IIGS, you cannot assign more than 256K to a RAM disk.

For serious development, you must have a hard disk, such as the Apple Hard Disk 20 SC. It is possible to run APW C from two 800K drives, but it requires considerable disk-swapping. If you use the C compiler with the assembler or the advanced linker, you will have considerable difficulty without a hard disk or at least three 800K drives.

Many developers find that an additional Apple II (not Apple IIGS) memory-expansion card is very useful. You can use the card for a large RAM disk on which you can place library files, compilers and assemblers, the linker, and utility programs. Since these programs are loaded into memory from disk each time they are used, placing them on a RAM disk can speed up the system's operation during program development.

❖ *Note:* If you haven't yet read the Preface, go back and read it now. In addition to providing a list of the manuals you'll need to develop programs for the Apple IIGS, it explains the layout of this book, the relationships of the books in the Apple IIGS Technical Library suite, and the conventions used to describe commands in this book.

The APW C disk contains the files shown in Appendix B. Use the index of this manual to get more information on any of these files. To examine the contents of your APW C disk, boot the disk, type `CAT` and press Return. To examine the contents of a subdirectory, include the pathname of the subdirectory; for example, to obtain a listing of the files in the subdirectory `/APWC/LIBRARIES`, use the following command:

```
CAT /APWC/LIBRARIES
```

To obtain a listing of all files in the volume `/APWC`, use the command

```
FILES +L +R /APWC
```

This command prints the contents of all directories in the volume and the files in each directory, with information about each file.

APW C concepts

This section describes a variety of features and concepts that you must understand in order to write application programs for the Apple IIGS computer. While some of these concepts may be familiar to you from your work with other computers, you must still be familiar with the way in which they are implemented on the Apple IIGS to get the most out of the Apple IIGS Programmer's Workshop, and to use the operating system and the memory of the Apple IIGS effectively.

Relocatable load files

The Apple IIGS Programmer's Workshop deals with three fundamental types of files: **source files,** object files, and load files. Source files are ASCII files consisting of the text of your program, and follow the conventions of a particular programming language; object files and load files are binary files conforming to the Apple IIGS object module format (OMF) defined in Chapter 7 of the *APW Reference.*

A C source file consists of C statements, preprocessor directives, function definitions and declarations, and so forth, together with variable declarations, which may include initialized data. In the source code, each specific function, variable, data structure labelled with a name. You can refer to the name in another part of the program: for example, you execute a function by using its name in a statement. A name or label of code or data used in this way is referred to as a **symbolic reference** (that is, a *symbol* that can be *referenced* or referred to). In high-level programming languages like C, symbolic references are usually the only means available to jump from one place in a program to another.

C uses a special kind of source file—a **header** or **include file**—containing code shared by many programs: for instance, lists of constants or interfaces to libraries. The header file is named in an `#include` statement in your source file, and the C compiler copies the header file in place of the `#include` statement before doing the actual compilation.

In assembly language, it is possible to specify actual locations in the computer's memory to which you want the program to jump: that is, you can write **absolute code.** The APW C Compiler only produces **relocatable code segments;** code segments that can be loaded into any location in memory. Note that such a program can be relocated only when it is loaded: once loaded, it can't be moved. (A program or block of code that can be moved from one location in memory to another while the program is running is called **position-independent.**)

The Apple IIGS system software and APW are both designed to support relocatable code.

When a source program is compiled, the compiler converts the source code into 65816 machine-language instructions, data declarations, and symbolic references. Before the program is actually run, the symbolic references must be **resolved;** that is, the routine being referenced must be found, and the reference must be replaced with code that the loader can use to relocate the code at load time. The program that resolves the symbolic references is called the **APW Linker.** (The linker gets its name from the fact that it can combine, or link together, several object files and library files to create a single executable load file.)

As shown in Figure 1-1, the conversion of a source file into 65816 machine language and data that are resident in memory is done in several steps:

1. The source code is compiled. The APW C Compiler first executes preprocessor directives, such as inserting include files, before compiling the source code and writing out one or more object files. Object files, then, consist of machine-language instructions and unresolved symbolic references to other routines.

   Your program can consist of several source files, each of which can be in any of the APW programming languages. Each source file is converted into one or more object files by the APW Assembler, the APW C Compiler, or any other APW compiler.

2. The object files are input to the APW Linker, which combines all of the object files into a single load file and resolves symbolic references. The linker verifies that every routine referenced is included in the load file. If there are any routines that the linker has not found when it has finished processing all of the object files, then it searches through any available library files for the missing routines. The linker removes symbolic references, and replaces them with entries in special tables it creates called **relocation dictionaries.** The load file consists of blocks of machine-language code that can be loaded directly into memory (called **memory images**), and relocation dictionaries that contain the information necessary to patch addresses into the memory images when the program is loaded into memory.

3. At program-execution time, the load file is loaded into memory by the System Loader. The loader calls the Apple IIGS Memory Manager to request blocks of memory for the load file, loads the memory images, and uses the relocation dictionaries to patch the actual memory addresses into the machine-language code in memory. The entire load file is not necessarily loaded into memory at one time; all OMF files are divided into **segments,** which can be processed independently. OMF-file segmentation is a fundamental Apple IIGS concept, which is discussed in the next section.

   The Memory Manager is the Apple IIGS tool set that allocates blocks of memory as needed and keeps track of which blocks of memory are available.

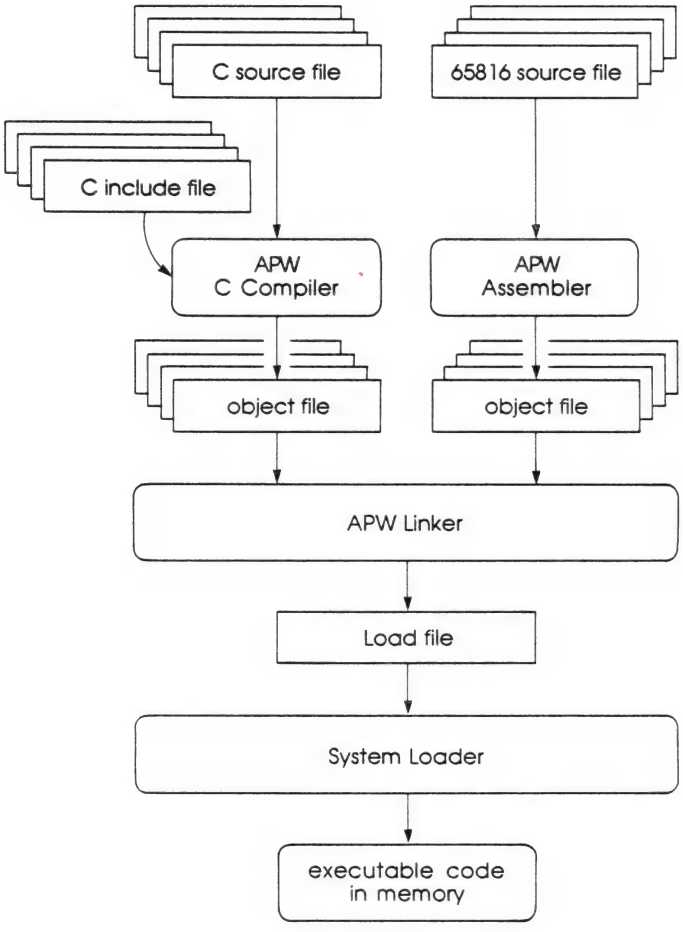


Figure 1-1
Creating an executable C program on the Apple IIGS

Program segmentation

In general, any computer program that consists of more than a few lines of code contains one or more subroutines; you may also segregate large blocks of data into separate parts of the program.

In APW C, each subroutine (called a **function**) is translated into a segment in the source file: the function name is the segment name. As illustrated in Figure 1-2, when you compile a program, each source-code segment is translated into one **object segment.**

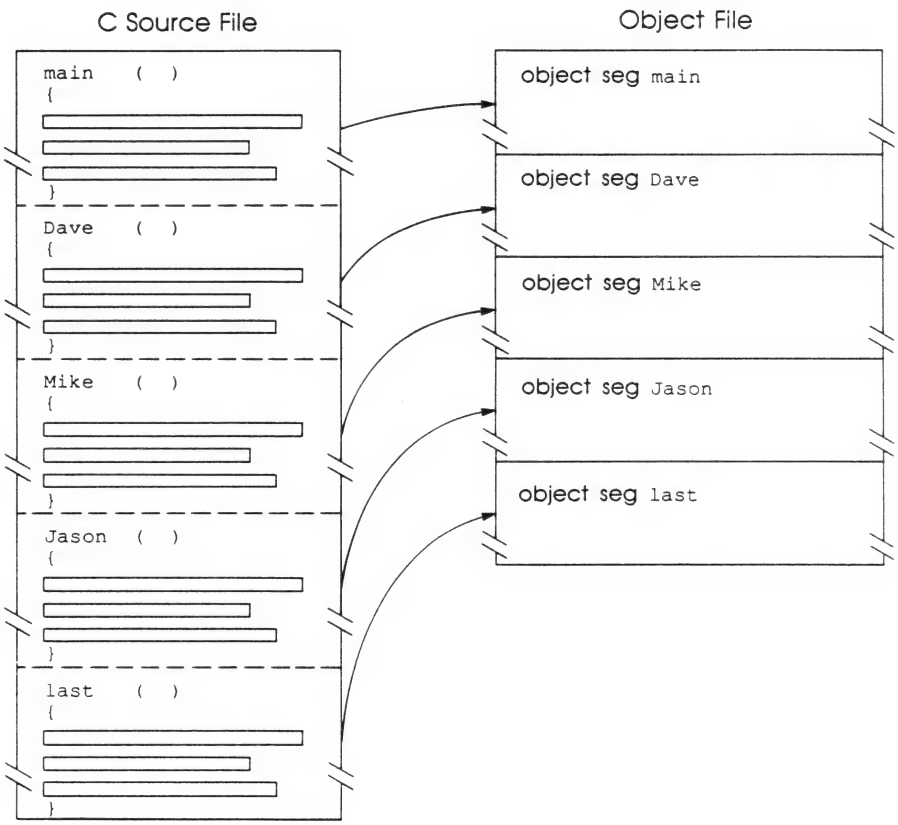


Figure 1-2
Creating object segments in source code

The object segment is the smallest linkable unit. For example, it can be selected from an object file for independent linking with a `LinkEd` command. Some compilers also can compile a segment (function) independently, this is called **partial compilation**.

❖ *Note:* The APW C Compiler does not perform partial compilation. If you request a partial compilation, the entire file will be compiled.

In addition to creating one code segment per function compiled, the APW C Compiler also creates two **data segments** for each object file created (that is, for each source file compiled). These segments are used for storing any global variables declared in the corresponding source file. Global scalar variables are stored in a segment called `~globals`, and global array and structure variables are stored in a segment called `~arrays`. Although this segmentation scheme means that each file will have the symbols `~arrays` and `~globals` defined, they are flagged as private symbols, which indicates they can only be accessed from within the object module they are contained in. The symbols for the variables themselves contained with the segments, are public. The compiler needs to generate two different data segments for the two different types of variables becauses it uses two different kinds of addressing—16-bit and 24-bit, respectively—to access them. Chapter 4 discusses the general implications of the code-generation memory model, as well as the implications for use with the advanced linker.

Apple IIGS load files also are segmented. Each load-file segment can incorporate any number of object-file segments. You can use a LinkEd command file to create load segments and to specify which object segments go in each load segment. Alternatively, APW C lets you specify load-segment names in the source code by using the `segment` command. If you do not use a LinkEd file, the linker places all code segments with the same load-segment name into the same load segment. The data segments `~globals` and `~arrays` are automatically identified as belonging to load segments of the same name; these must be collected into their own load segments so that the system loader can be assured of loading the `~globals` segment within a single bank as required by the code-generation model, and so that the data segments can be reloaded independently of the code when a program is restarted. Again, the linker does this automatically unless you use a LinkEd file to control your link. Use of source-file load-segment names are illustrated in Figure 1-3.

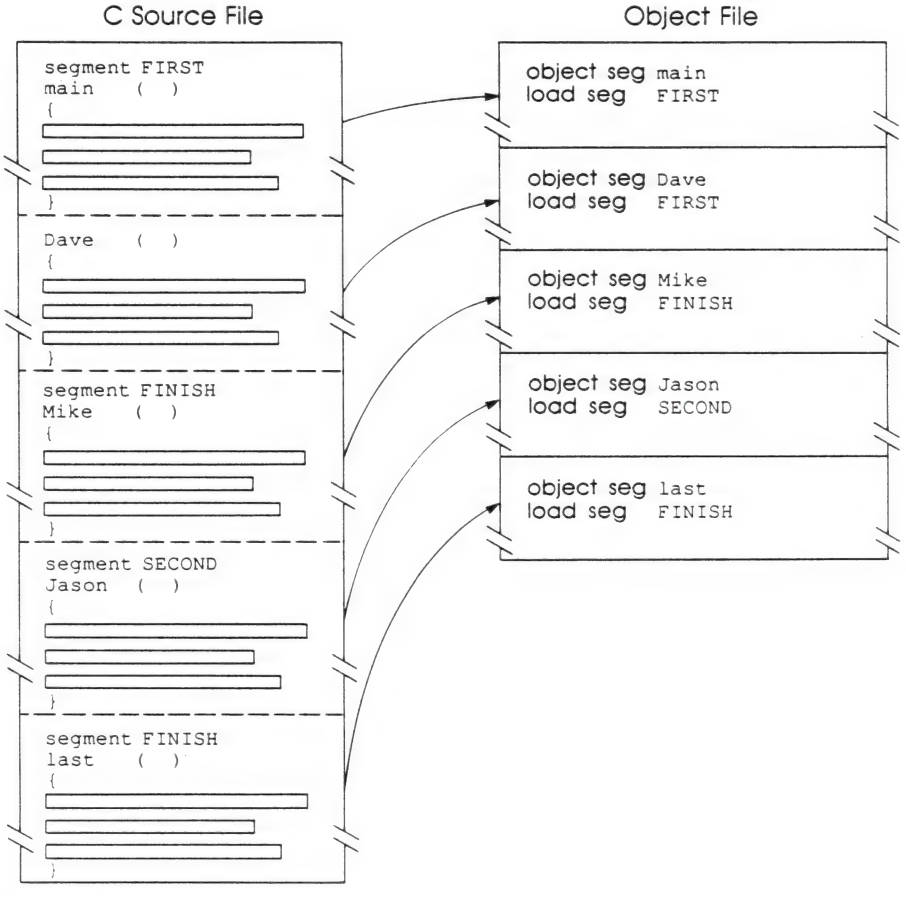


Figure 1-3
Assigning load segments in source code

The relationship of object segments to load segments is illustrated in Figure 1-4.

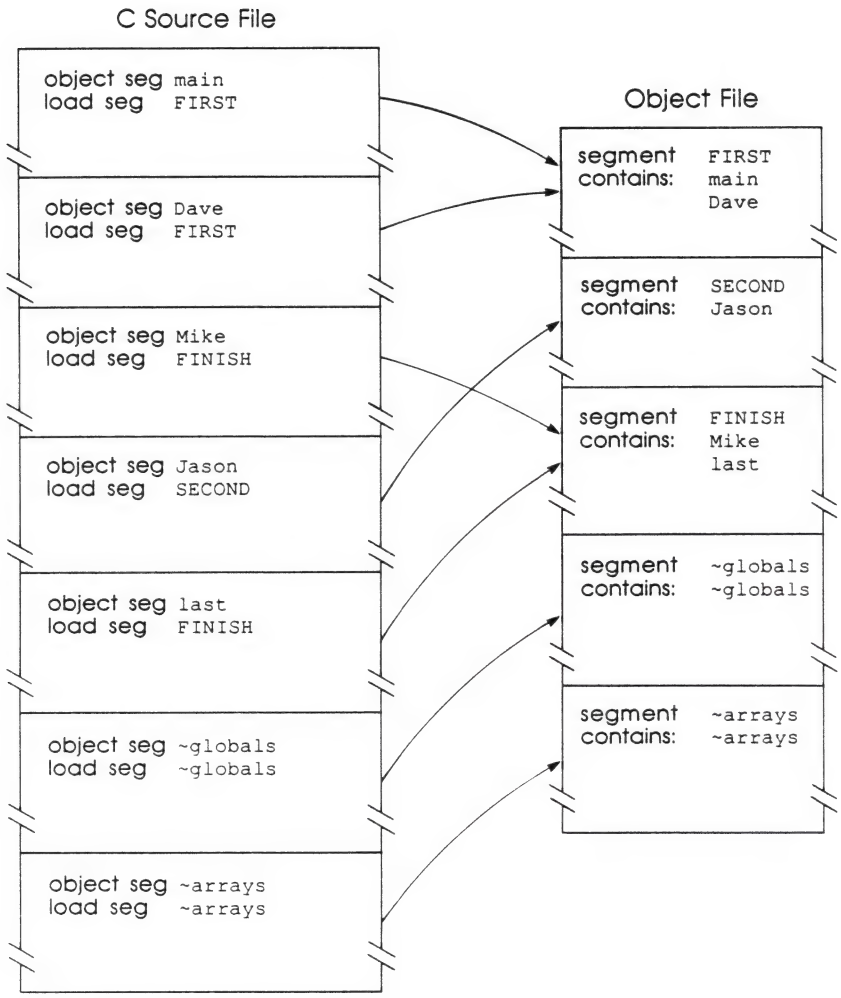


Figure 1-4
Relationship between object segments and load segments

Every OMF file consists of one or more segments, each comprising a **segment header** and a **segment body.** The segment header is divided into fields described in "Segment Header" in Chapter 8 of the *APW Reference*.

A load-segment header contains the name of the segment; an object-segment header contains the name of the segment and the name of the load segment into which it goes. The linker uses the name of the object segment in resolving function references; also, you specify the names of object segments when using the advanced linker to extract specific segments for linking (see "Using the Advanced Linker" in Chapter 5 of the *APW Reference*).

Each segment in a program must have a unique object-segment name: in APW C, each function is compiled to a separate object segment, whose name is the function name. Each object segment is also assigned a load-segment name. As illustrated in Figure 1-4, APW C lets you assign your own load-segment name to an object segment. Any number of object segments can have the same load-segment name. The standard linker places all object segments that share the same load-segment name into the same load segment (as long as they will fit into 64K).

For example, suppose your object file contains the following segments:

0. Object Segment Name: `main`
   Load Segment Name: `FIRST`
1. Object Segment Name: `Dave`
   Load Segment Name: `FIRST`
2. Object Segment Name: `Mike`
   Load Segment Name: `FINISH`
3. Object Segment Name: `Jason`
   Load Segment Name: `SECOND`
4. Object Segment Name: `last`
   Load Segment Name: `FINISH`
5. Object Segment Name: `~globals`
   Load Segment Name: `~globals`
6. Object Segment Name: `~arrays`
   Load Segment Name: `~arrays`

When the standard linker processes this file, object-segment names `main`, `Dave`, `Mike`, `Jason`, and `last` are treated as references that must be resolved. Object segments `main` and `Dave` are placed in the same load segment, which is named `FIRST`; object segments `Mike` and `last` are placed in the same load segment, which is named `FINISH`; and object segment `Jason` is placed in a separate load segment, which is named `SECOND`. Additionally, the object segment `~globals` is placed in the load segment `~globals`, and the object segment `~arrays` is placed in the load segment `~arrays`.

On the Apple IIGS computer, no single block of code can occupy more than 64 K of contiguous memory. To load a larger program than that, you must split the program up into two or more load segments. When most of memory is already in use, the loader may be able to load a program that is divided into several small load segments even if the same program in one or two load segments wouldn't fit. The Apple IIGS Memory Manager takes care of assigning each segment to a memory block; the System Loader keeps track of where in memory the segment has been loaded and patches intersegment calls in each segment as it is loaded.

Dynamic segments

On the Apple IIGS computer, the combination of load segments, the System Loader, and the Memory Manager makes possible the creation of **dynamic segments.** The loader and memory manager can load a dynamic segment automatically during program execution simply by calling a function contained within the dynamic segment; if the segment is not currently in memory, the loader will load it automatically. A dynamic segment that is not needed at a given time can be removed, freeing the memory used to allow room in which to load another dynamic segment or, indeed, for any other purpose. In additional, the loader and Memory Manager actually purge a dynamic segment from memory only if the memory is needed for something else; otherwise, the segment remains in memory and need not be reloaded the next time it is called, even if the user has "unloaded" it.

A segment that is not dynamic is **static.** A static segment is loaded at program boot-time and is not unloaded or moved during execution. The first segment of any program that is loaded is static; any other segments may be static, but (especially for large programs) the system will be more memory efficient if all infrequently used segments are dynamic. These dynamic segments may make development of large applications for smaller memory configurations practical. To specify that a load segment is dynamic, you must use a `LinkEd` command or specify the **`dynamic`** option to the `segment` command.

Library files

Library files contain routines that are useful to many different programs. On the Apple IIGS, all library files are in object-module format, regardless of the language of the source file. An Apple IIGS library file (ProDOS file type `$B2`) can therefore be used by a program written in any source language. Some languages, such as APW C, come with a set of library files used by that language. When the linker processes one or more object files and cannot resolve a symbolic reference, it assumes that the reference is to a segment in a library file. If you use the standard linker, it automatically searches all library files in the APW library prefix (`2/`). (If you use a `LinkEd` command file, then the advanced linker searches only the library files that you specify.) Unless you are using the advanced linker, you do not even need to know the names of the library files in order to use them: the standard linker automatically finds the files and extracts the segments it needs.

You can create your own library files from one or more object files by using the MakeLib APW utility program. Figure 1-5 illustrates the process of creating a library file. You specify one or more object files to be included in the library file. MakeLib concatenates the files and creates a special segment at the beginning of the file called the **library dictionary segment.** The library dictionary segment is the first segment of a library file, and contains the names and locations of all **global symbols** in the file. (A global symbol is a label in one segment that can be referenced in another segment, as opposed to a **local symbol,** which can be used only within the segment in which it is defined.) The linker uses the library dictionary segment to find the segments it needs.

The library dictionary segment allows the linker to search a library file for global symbols much more rapidly than it can search an object file. Consequently, the linker will search a library dictionary segment more than once, if necessary, to find segments referenced by other segments in the library file. Therefore, the sequential order of the segments in a library file is therefore not important. However, if you were to use several library files, the order in which the files were searched *would* be important. If the linker first searched file `A` and then file `B`, for example, it could resolve a reference made in file `A` to a global symbol in file `B`, but could not resolve a reference made in file `B` to a symbol in file `A`. It is for that reason that MakeLib allows you to include several object files in a single library file.

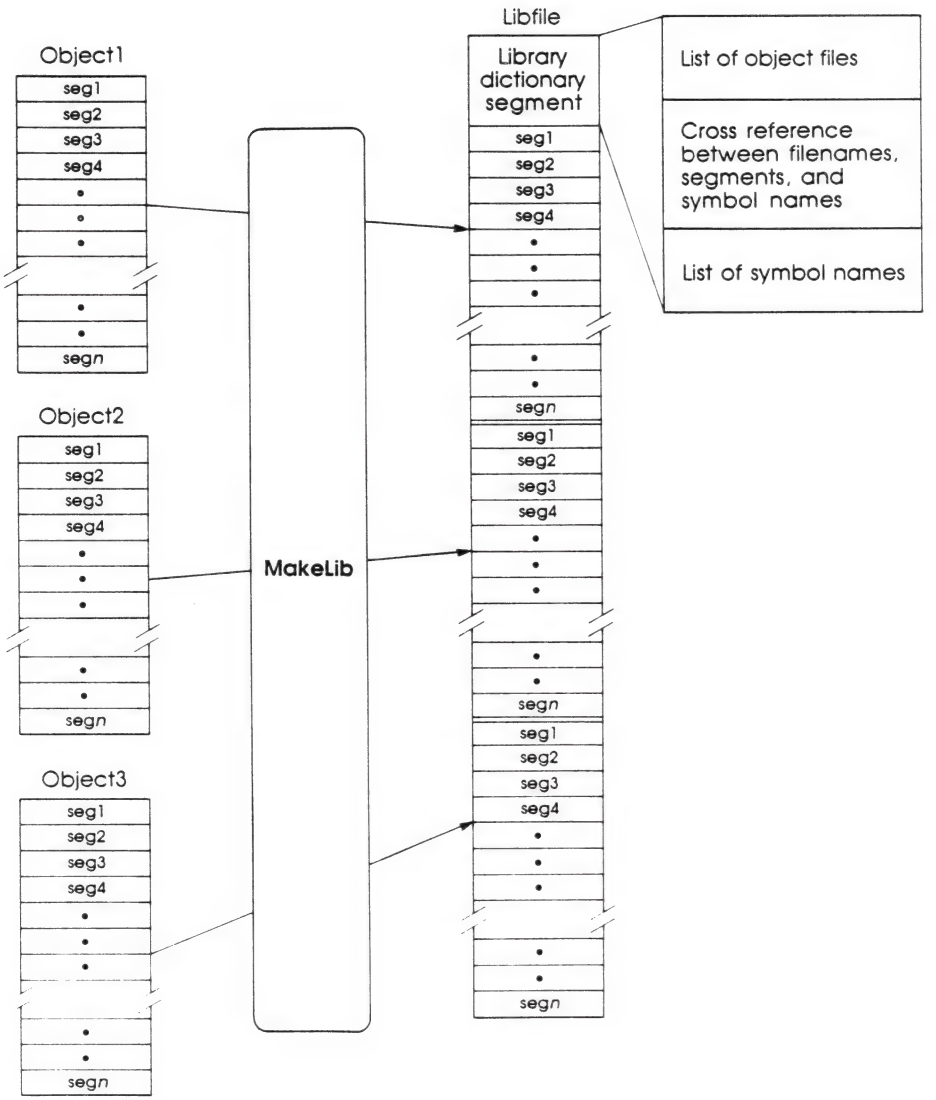


Figure 1-5
Relationship between object files and library files

Program interactions

This section illustrates the interactions among the various programs in the Apple IIGS Programmer's Workshop by presenting a typical sequence of procedures and events. For this purpose, this manual assumes that you are developing an application written mostly in C, with some routines written in 65816 assembly language. In this section, only the sequence of operations is listed; Chapter 3 provides an actual example of the sequence described here. The process described in this section is illustrated in Figure 1-6. See the *Apple IIGS ProDOS 16 Reference* for a complete description of the program-load process.

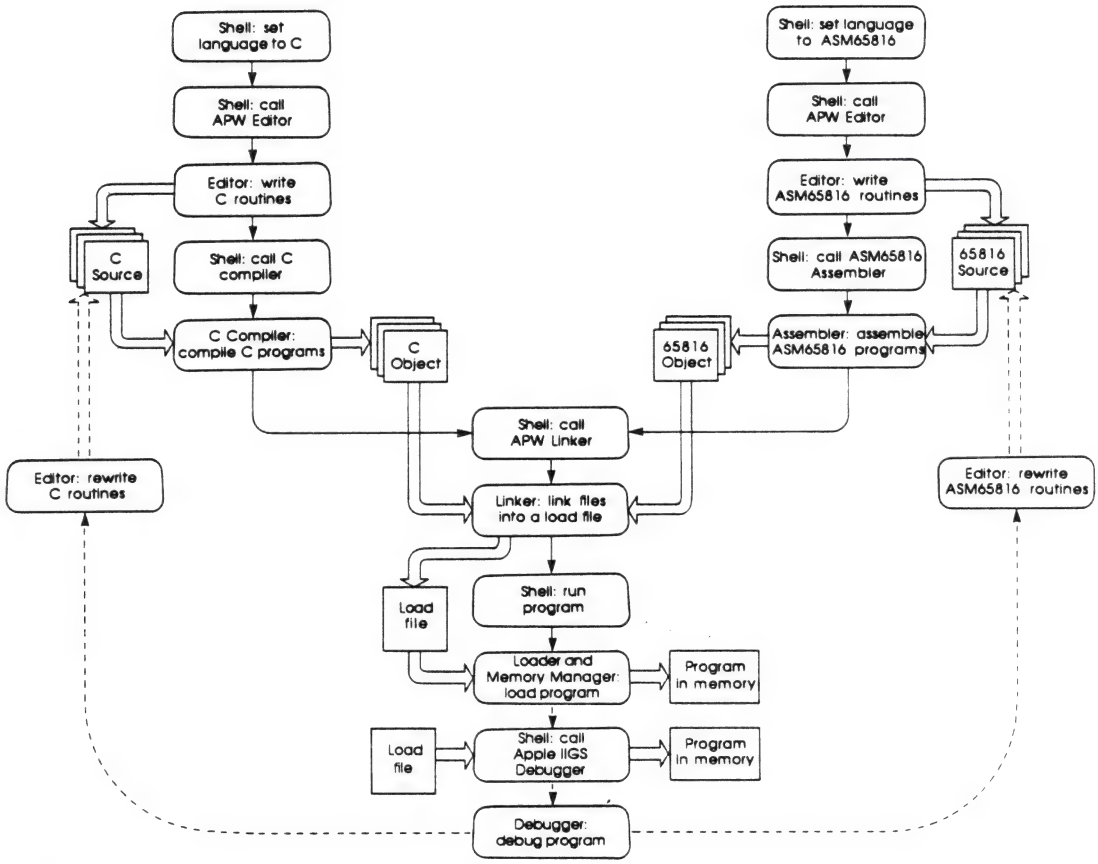


Figure 1-6
Program interactions

1. Using an APW Shell command, set the **current language** for APW to CC. (Every APW file has an APW language type; if you open a new file, it is given the current APW language type.)
2. Call the APW Editor and open a new file.
3. Use the editor to write the C language routines. You can divide the program among as many files as you wish. You do not have to return to the shell between files; you can save one file and open another within the editor. In APW C, you can use the `segment` command to specify which object segments go in which load segments. Until you use a shell command to change it, or until you open a non-C file, the current language remains CC.
4. Quit the editor, change the current language to `ASM65816`, call the editor, and open a new file. You can divide the 65816 assembly-language routines among as many files and as many segments per file as you wish. The APW Assember lets you specify which object segments go in which load segments. Make the assembly-language routines relocatable; that is, use no absolute addresses—use labels and relative addressing only.

   If you have used macros in your assembly-language program, you can run the MacGen utility to generate a custom macro file for the program.

Until you use a shell command to change it, or open a non-assembly-language file, the current language remains ASM65816.

5. Quit the editor, call the APW Assembler to assemble the 65816 assembly-language routines, and call the APW C Compiler to compile the C routines. You can use the same command for both.
6. Use the APW Linker to link the object files into a load file. Normally, you can use the standard linker to link the program. The standard linker places all object segments with the same load-segment name into a single load segment.
   a. If you want to change load-segment assignments, or if you want to respecify dynamic load segments, you must use the advanced linker. Write a LinkEd file like a language source file: first set the system language to LINKED and then use the editor to write the file.
7. Run the program by typing the name of the load file and pressing Return. (You can also automatically execute a program after linking by using the CMPLG command.) When a program is run on the Apple IIGS, the following events occur:
   a. The System Loader loads the first segment into memory (calling the Memory Manager to request the block of memory it needs). This segment is static; that is, it remains in memory during the execution of the program. The loader uses the relocation dictionary of the segment to relocate the code to its present location in memory.
   b. The loader loads all other static segments into memory, relocating them as necessary.
   c. The loader passes control of the system to the program, and the program begins execution.
   d. When the program encounters a reference to a subroutine in a dynamic segment, control is returned to the System Loader through the jump table. If the segment is already in memory, the loader transfers control to the segment. If not, the loader uses the jump table to locate the load file, segment, and offset of the subroutine, loads the segment into memory, and transfers control to the segment. The System Loader creates and maintains a table (the **Memory Segment Table**) to keep track of all the segments in memory.

   ❖ *Note:* If the program does not run correctly, you can use the Apple IIGS Debugger (available as a separate product from APDA) to step through or trace the code, to insert breakpoints, to disassemble the machine code, and to examine the contents of registers and memory locations. You can modify the code in memory and rerun the program until the bug is fixed.
8. Correct the source code and recompile (or reassemble) the program.
9. Relink the program and rerun it.
10. When the program is completely debugged, you can use the CRUNCH command to compress the files created by partial assemblies into two object files, and then link the program one last time. Using CRUNCH is optional: if you have performed several partial assemblies, compressing the object files speeds up the link process.

Using the APW C libraries

APW C programs can use the Standard C Library, the Apple IIGS Toolbox, the APW Shell, and ProDOS to talk to the Apple IIGS hardware. All of the interface code to make these calls is in the file `CLIB` which is installed in the APW library prefix (`2/`). (Any header files containing declarations needed to make the calls are installed in the CINCLUDE directory in the library prefix.) Figure 1-7 shows how these libraries interact. Your application can make calls to the Standard C Library, the APW Shell, the Apple IIGS Toolbox, or ProDOS. The Standard C Library contains a number of high-level routines familiar to C programmers, which deal with file handling, memory management, and so on. The Standard C Library in turn calls the Toolbox or ProDOS. You can also make calls to the APW Shell. The shell intercepts the call: if it is a ProDOS call, the shell passes it through unchanged; if it is a shell call, the shell makes ProDOS calls or talks to the hardware directly in order to execute it.

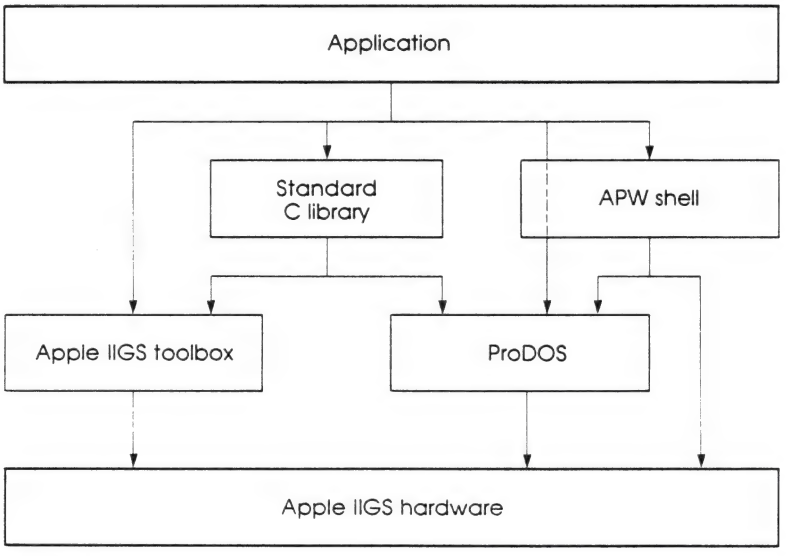


Figure 1-7
APW C library interactions

Chapter 2

Using the APW C Compiler

This chapter describes how to use the APW C Compiler. The first section, "Installing APW C," tells you how to install APW C in both hard-disk and 3.5-inch disk systems. The second section, "Running APW C on 3.5-Inch Disks," tells you how to run APW C on a 3.5-inch disk. The third section, "Writing and Running a Sample Program," leads you through a sample session, giving you a fast way to become acquainted with compiling, linking, and executing a program. The third section, "The APW C Compiler," discusses the compilation process. The fourth section, "C Compiler Shell Commands," describes the shell commands you'll use when working with the C compiler. The fifth section, "Files for Compiling and Linking," tells how to use the various files used in building a program.

Installing APW C

Before you can follow any of the procedures described in this chapter, you must install APW C. First back up your disks, then install APW, and then install C, as described in this section.

Backing up your APW C disk

It is important to make a backup copy of your APW C disk and to run APW from the copy only. Keep the original disk in a safe place so you can make a new copy if something happens to the copy you've been using.

You back up the APW C disk in the same way you back up the other APW disks. If you have not backed them up yet, back up all three now: APW, APW Assembler, and APW C.

You can make a copy of your APW C disk by using any disk-copy utility you prefer, or you can use APW commands to do the job, as explained in Chapter 3 of the *APW Reference*.

Important:

You must give your copy of the `/APWC` disk the volume name `/APWC`, or the hard-disk installation procedure will not work correctly. Similarly, your backup of each of the APW disks must have the same volume name as the original disk.

Installation

This section assumes you have already installed APW (version 1.0 or later), as described in Chapter 2 of the *APW Reference*.

To install APW C, launch APW; then type the following command:

```
INSTALL /APWC
```

This command copies the necessary files. This process will take several minutes.

You now have APW C installed.

Installing APW C may replace the files SYSCMND, LOGIN, and SYSTABS in the APW/SYSTEM subdirectory. If you have customized any of these files, you should rename them before installing the new APW, and then either edit or replace the new versions of the files as appropriate.

Running APW C on 3.5-inch disks

You need at least two 800K disk drives to use APW: one to hold the /APW disk, and one to hold either the /APWC disk or a disk containing only the files you are working on.

Important

Do *not* run APW C from the original product disks. Make copies of your APW disks for everyday use, and put the original disks in a safe place.

The /APW disk contains the Apple IIGS Program Launcher and a fully functional APW system, including the APW Assembler. This disk lacks only the help files and some of the APW utility programs. The /APWU disk contains a full set of utility programs plus the help files for all the APW commands. The /APWC disk contains the C compiler, the linker, and the libraries you need for C.

Launch APW as before and place the /APWC disk in the second disk drive. To cause APW to look on the /APWC disk for the C compiler and linker, enter the following command:

```
MC
```

If you want to use the assembler, enter the following command:

```
UMC
```

To go back to the C compiler, enter the following command:

```
MC
```

The directory that is assumed when you do not specify a prefix in a pathname is called the **current prefix.** If the /APW disk is in your first disk drive and all of your program files are on the disk in the second disk drive, you may wish to set the system to use a directory on your program-file disk as the current prefix. Use the APW Shell's PREFIX command to change the current prefix. For example, if your programs are in a subdirectory called /APWC/MYPROGS/ in the second disk drive, type the following command and press Return:

```
PREFIX /APWC/MYPROGS
```

After you have set the current prefix to that of your program disk, you need not include the prefix in pathnames when executing commands. For example, if the current prefix is /APWC/MYPROGS/, you could use the following command to obtain a directory listing of the subdirectory /APWC/MYPROGS/CSOURCE/:

```
CATALOG CSOURCE
```

- *Note:* Do *not* include a slash (/) before the pathname when you omit the current prefix from a pathname, or APW will look for a volume of that name. For example, if you typed CATALOG/CSOURCE in the preceding example, you would get the message "Volume not found."

Prefixes used by APW are discussed in detail under "Using Prefix Numbers" in Chapter 2 of the *APW Reference.*

Keep the /APW disk in the first disk drive while you are running APW so that the system can have access to the APW programs on that disk.

Each time you start APW, it looks for a file named LOGIN in the APW system prefix (/APW/APW/SYSTEM/LOGIN on the /APW disk, for example). The LOGIN file should have an APW language type of EXEC (see "Listing a Directory" in Chapter 2 of the *APW Reference*). You can include any valid APW command in this file. If APW finds a LOGIN file, it executes the file before doing anything else.

You can use a LOGIN file to set system defaults (such as the printer slot), to set the current prefix, to read a command table containing command-name aliases, or even to execute commands or utility programs.

You need not have a LOGIN file in your system; if there is no LOGIN file, APW uses default settings for system parameters.

Writing and running a sample program

Here is the way to write, compile, link, and run a trivial sample program.

Writing the sample program

First set the current language to C by typing CC and pressing Return. Now create a new file named SHE.SELLS by typing EDIT SHE.SELLS and pressing Return.

❖ *Note:* If you ever get the error message "ProDOS: File not found", make sure you've typed the command correctly. If you had typed ED rather than EDIT, for example, the APW Shell will give you this message because it knows no command named ED and can't find any file with that name. (ProDOS is not complaining that it couldn't find a file named SHE.SELLS.)

When you are in the APW editor, type a program; for example, type

```
main()
{
        printf("She sells C shells by the C shore.\n");
        return 0;
}
```

Press Apple-Q or Control-Q, and then type S, to save the program; then type E to exit the editor.

Note that APW does not require the usual C filename extension .c, because APW uses a unique file type for source files of each language. You can end a filename with .c, but the APW C Compiler regards the .c as part of the name, rather than as an extension. In particular, when forming an object filename, the compiler appends an extension to the .c, rather than replacing it. Using .c on a source filename can be confusing, as some object filenames have a .c extension.

Compiling and linking the sample program

To compile your program, use the COMPILE command; to link it, use the LINK command.

For example, to compile and link SHE.SELLS, creating an object file C.SHELLS.ROOT and a load file C.SHELLS, type the following commands and then press Return:

```
COMPILE SHE.SELLS KEEP=C.SHELLS

LINK 2/START C.SHELLS KEEP=C.SHELLS
```

Running the sample program

To run your program under the APW shell, type C.SHELLS and press Return. You will see `She sells C shells by the C shore.` on the screen.

A more interesting sample program, written in both C and assembly language, is in Chapter 3.

The APW C Compiler

This section discusses the compilation process, the way that compilation is suspended or aborted, and error messages.

The compilation process

The APW C Compiler is a *one-pass compiler.* In one pass, the compiler resolves preprocessor macros, scans the source files, and generates code into a code buffer; it then writes the code to an object file. Each C function is assigned to a separate object segment: the object-segment name is the function name. The default load-segment name is MAIN.

You can use the segment command to assign an object segment or group of segments to a load segment, which can be either static or dynamic.

No listing is printed. If requested, the compiler prints progress information and error messages to the screen.

Object-code output is in object module format. Each APW language outputs object code in object module format, allowing you to link together subroutines written in different languages. Object module format is discussed in detail in Chapter 8, "File Formats," of the *Apple IIGS Programmer's Workshop Reference.*

If there are no more subroutines to compile, the compiler returns control to the shell. Depending on the command you used to invoke the compiler, the shell either passes control to the linker or returns with the shell prompt. If called, the linker relocates the object modules produced by the compiler to resolve **global labels,** and writes out an executable binary file.

Suspending or canceling the compilation

You can suspend the compilation process by pressing any key. Pressing any key again causes compilation to resume. To cancel the compilation process, press Apple-Period (⌘-.).

C compiler error messages

If the C compiler detects an error in the source code, by default it returns to the editor, with the cursor on the offending line and with an explanatory error message at the bottom of the screen. The default behavior can be overridden by using the `-E` option to the `COMPILE` command.

If the default error behavior is overridden, the compiler prints an error message on the screen. Each error message includes the source file name, the line number, and the text of the offending line of code. In other cases, the compiler prints a warning message rather than an error. Error messages can be redirected, as explained in "Redirecting Input and Output" in Chapter 3 of the *APW Reference*. If no errors or warnings are detected, the compiler runs without comment.

C compiler shell commands

This section discusses the commands you'll use most often when working with the C compiler. With these commands, you can perform the following tasks:

- edit new and existing files
- compile, link, and execute your program
- make a library file
- debug your program

Editing a source file

You will need three shell commands when you edit a new or existing source file:

`CC`	Change the default language to C
`EDIT` *filename*	Edit a new or existing file
`CHANGE` *filename* `CC`	Change the type of an existing file to C source file

The `CC` command sets the default language to C. Any new files you create with the editor will then automatically get the appropriate type for a C source file. The `EDIT` command edits an existing file or creates a new file. The `CHANGE` command changes the type of a file from one language to another. Doing this is useful if you have imported an ASCII file from some other implementation of C, such as MPW, and the file type is not set for APW C; or if you had created a C source file when the default language was not set to C.

Compiling and linking a program

You'll need two commands when compiling, linking, and running your program:

COMPILE	Compile a program
LINK	Link a program

In its simplest form, the COMPILE command compiles the source file, but saves no object file: it simply verifies the program's correctness. To create an object file, use the KEEP option or the KEEPNAME shell variable, both described later in this chapter. The COMPILE command is a synonym of the ASSEMBLE command. These commands can be used interchangeably to compile or assemble programs. Synonymous commands have the same options, but one language processor may ignore options that another recognizes. For example, the C compiler ignores the +L|-L and +S|-S options.

Some other commands are useful:

CMPL	Compile and link a program
CMPLG	Compile, link, and execute a program
RUN	Compile, link, and execute a program

CMPL is a synonym of ASML, and CMPLG and RUN are synonyms of ASMLG.

Command notation

The following notation is used to describe commands:

UPPERCASE — Uppercase letters indicate a command name or an option that must be spelled exactly as shown. The shell is not case-sensitive; that is, you can enter commands in any combination of uppercase and lowercase letters. Segment names *are* case-sensitive. In case-sensitive languages like C, segment names must be entered exactly as they appear in the source code. Segment names in case-insensitive languages must be entered in uppercase.

italics — Italics indicate a variable, such as a filename or address.

prefix — This parameter indicates any valid directory pathname or partial pathname. It does *not* include a filename. If the volume name is included, *prefix* must start with a slash (/); if *prefix* does not start with a slash, then the current prefix is assumed. For example, if you are copying a file to the subdirectory SUBDIRECTORY on the volume VOLUME, then the *prefix* parameter would be /VOLUME/SUBDIRECTORY/. If the current prefix were /VOLUME/, then you could use SUBDIRECTORY for *pathname*

The device numbers .D1, .D2, ...D*n* can be used for volume names. If you use a device number, do not precede it with a slash. For example, if the volume VOLUME in the example given earlier were in disk drive .D1, then you could enter the *prefix* parameter as .D1/SUBDIRECTORY/.

filename — This parameter indicates a filename, *not* including the prefix. The unit names .CONSOLE and .PRINTER can be used as filenames.

pathname This parameter indicates a full pathname, including the prefix and filename, or a partial pathname, in which the current prefix is assumed. For example, if a file is named `FILE` in the subdirectory `DIRECTORY` on the volume `VOLUME`, then the *pathname* parameter would be `/VOLUME/DIRECTORY/FILE`. If the current prefix were `/VOLUME/`, then you could use `DIRECTORY/FILE` for *pathname*. A full pathname (including the volume name) must begin with a slash (/); do *not* precede *pathname* with a slash if you are using a partial pathname.

The unit names `.CONSOLE` and `.PRINTER` can be used as filenames; the device numbers `.D1, .D2, ... .D`*n* can be used for volume names.

| A vertical bar indicates a choice. For example, `+L|-L` indicates that the command can be entered as either `+L` or as `-L`.

A|B An underlined choice is the default value.

[ ] Parameters enclosed in square brackets are optional.

... Ellipses indicate that a parameter or sequence of parameters can be repeated as many times as you wish.

The following pointers will help you use the APW Shell command interpreter:

- You must separate the command from its parameters by one or more blanks.
- You can use the Right-Arrow key to expand command names as described in "Entering Commands" in Chapter 2 of the *APW Reference,* and you can use the Up-Arrow and Down-Arrow keys to scroll through previously entered commands.
- There are no abbreviations for command names (unless you define your own with `ALIAS` or by changing the `SYSCMND` file).
- All commands and parameters (except for segment names) can be entered in any combination of uppercase and lowercase characters.
- For case-sensitive source languages like C, segment names must be entered exactly as they appear in the source code. (For case-insensitive source languages like assembly language, segment names must be entered in uppercase.)
- When a parameter in a command line conflicts with a source-code command, the command-line parameter takes precedence. When neither a source-code command nor a command-line parameter has been used, the default parameter is used.
- If you fail to enter a required parameter, you are prompted for it.
- Any of these commands can be placed in an Exec command file for automatic execution; Exec files are described in "Exec Files" in Chapter 3 of the *APW Reference.*

The APW Shell and APW C Compiler recognize the commands listed here. The options for each command are described after the command.

CC

The CC command sets the shell default language to APW C. Any file the APW Editor creates while this command is in effect will have the proper file type for a C source file. (This command is described in "Command Descriptions" in Chapter 3 of the *APW Reference*.)

CHANGE

CHANGE *filename* CC Change the type of an existing file to C source file

The CHANGE command changes the file type of an existing file named *filename* so that APW will recognize it as a C source file. This command is useful when you have imported a C source file from another development system, such as MPW, that does not identify the language of a source file by a unique file type. (This command is described in "Command Descriptions" in Chapter 3 of the *APW Reference*.)

CMPL

CMPL [*option* ...] *file1* [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [CC=(*option* ...)
[*language2*=(*option* ...)] [...]] [...]]

The CMPL command compiles (or assembles) and links one or more source files and links one or more object and library files. The APW Shell checks the language of the source file and calls the appropriate compiler or assembler. If the maximum error level returned by each assembler or compiler is less than or equal to the maximum allowed (0 unless you specify otherwise with the MERR directive or its equivalent in the source file), the standard linker is called to link the resulting object files and any other object and library files named on the CMPL command line. The linker is described in Chapter 5 of the *APW Reference*.

The CMPL command is an alias for ASML.

The CMPL command is described fully in Chapter 3 of the *APW Reference*.

For examples and discussion of the use of the CMPL command, see Chapter 3 of this manual and "Compiling (or Assembling) and Linking a Program" in Chapter 2 of the *APW Reference*.

CMPLG

CMPLG [*option* ...] *file1* [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]

The CMPLG internal command compiles (or assembles) one or more source files, links one or more object and library files, and runs the resulting load file. The function of CMPLG is identical to that of the CMPL command—except that, once the program has been successfully linked, it is executed automatically. See the description of the CMPL command for a list of options and a description of the parameters.

The ASMLG and RUN commands are aliases for CMPLG.

COMPILE

COMPILE [*option* ...] *file1* [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]

The COMPILE internal command compiles (or assembles) one or more source files. You can use the LINK command or a LinkEd file to link the object files created by the COMPILE command. The APW Shell checks the language of the source file and calls the appropriate compiler or assembler.

The ASSEMBLE command is an alias for COMPILE.

The options that apply to APW C are described next; other options are described in Chapter 3 of the *APW Reference*.

❖ *Note:* Not all compilers or assemblers make use of all the parameters provided by this command (or by the ASSEMBLE, ASMLG, COMPILE, CMPL, CMPLG, and RUN commands, which use the same parameters). The APW C Compiler, for example, does not produce a listing or symbol table, and so ignores the +L|-L and +S|-S options. If you include a parameter that a compiler or assembler cannot use, it ignores the parameter: no error is generated.

If you include more than one source file or use #append directives to tie together source files in more than one language, then all parameters are passed to every compiler or assembler. Each compiler or assembler uses those parameters that it recognizes. The reference manual for a compiler or assembler contains a list of the options that it accepts.

❖ *Note:* Command-line parameters (those described here) override source-code options when there is a conflict.

Important

If you are using a LinkEd file to take advantage of advanced linker capabilities, do not use the CMPL command. Instead, use the COMPILE command to compile your program. You can process the LinkEd file automatically by appending it to the end of your program with an #append directive (or the equivalent), or you can process it independently with the ALINK command.

❖ *Note:* You can use #append directives (or the equivalent) to tie together source files written in different computer languages; APW compilers and assemblers check the language type of each file and return control to the shell when a different language must be called. See "Compiling (or Assembling) and Linking a Program" in Chapter 2 of the *APW Reference* for a description of the assembly and compilation process.

□ *option...* You can specify as many of the following options as you wish by separating the options with spaces.

  □ +E|-E If you specify +E, when the compiler terminates execution due to a fatal error, it calls the APW Editor. The editor displays the source file with the offending line on the fifth line on the screen (or as far down on the screen as possible, if the error is in one of the first four lines of the file). If you specify -E and a fatal error occurs, the compiler returns you to the shell's command line or to the Exec file that executed the command. The default for this option is +E when the command is executed from the command line, and -E when the command is executed from an Exec file.

- +L|-L The APW C Compiler ignores this option.
- +S|-S The APW C Compiler ignores this option.
- +T|-T If you select +T, any error causes the compile to terminate. If you omit this option or select -T, only fatal errors cause immediate termination of the compile. Note that if you select both +T and +E, any error causes the shell to call the APW Editor and display the offending line as the fifth line on the screen.
- +W|-W If you select +W, the compiler stops and waits for a key press when any error occurs, to give you the opportunity to read the error message and to decide whether to continue (that is, to continue the compilation in case of a nonfatal error or to call the editor in case of a fatal error). Press Apple-Period to halt execution, or press any character key or the space bar to continue. If you omit this option or select -W, execution continues without pausing when an error occurs.

- *file1 file2* ... The full pathnames or partial pathnames (including the filenames) of the source files to be assembled (or compiled). You may include as many source, object, and library files as you choose, but at least one of the files must be a source file. Separate the filenames with spaces.

  The source files do not all have to have the same APW language type. Note, however, that if you include a LinkEd file, it must be the last file listed. The reason is that once the advanced linker has been called by a LinkEd file, the linker is not called again regardless of how many source or object files follow the LinkEd file.

- KEEP=*outfile* You can use this parameter to specify the pathname or partial pathname (including the filename) of the output file. There must not be any spaces between KEEP and the equal sign (=).

  For a one-segment program, the assembler or compiler names the object file *outfile*.ROOT. If the program contains more than one segment, the assembler places the first segment in *outfile*.ROOT and the other segments in *outfile*.A. If this is a partial assembly (or several source files with different programming languages are being compiled), other filename extensions may be used; see "Partial Compilation or Assembly" in this chapter.

  If the assembly is followed by a successful link, the load file is named *outfile*.

  Keep the following points in mind regarding the KEEP parameter:

  - You can specify a default filename for object files by using the KeepName shell variable. Shell variables are described in "Variables" in Chapter 3 of the *APW Reference*.
  - To use the KEEP parameter with multiple source files, you must use one or more wildcard characters in the KEEP parameter.
  - Because ProDOS 16 does not allow filenames longer than 15 characters, you must be careful not to specify a filename in the KEEP parameter that will result in an output filename longer than 15 characters. For example, if you specify KEEP=%.OUT and the source filename is LONGNAME, the compilation will fail when the shell tries to open the file LONGNAME.OUT.ROOT, which has 17 characters.
  - If *object* files with the root filename *outfile* already exist, they are overwritten without a warning when this command is executed.

- NAMES=(*seg1 seg2 ...*) This parameter is ignored by the APW C Compiler, which always compiles all C source files listed on the command line. APW Assembler uses this parameter for partial assembly.
- CC=(*option ...*) This parameter allows you to pass parameters directly to the APW C Compiler. Between the parentheses, insert one or more of the options listed next. Note that the APW shell does no error checking on this string before passing it through to the compiler or the assembler. This parameter is a special case of the *language1*=(*option ...*) parameter.

  ❖ *Note:* No spaces are permitted immediately before or after the equal sign in this parameter.

  This option's options are as follows.

  - -D*name=value* This option defines *name* as if a `#define` had occurred at the top of the file. The variable *name* is given the value *value*. No spaces are permitted immediately before or after the equal sign in option parameter.
  - -D*name* This option (a special case of the one just given) defines *name* as if a

    `#define` *name* `1`

    had occurred at the top of the file. The variable *name* is given the value 1. No spaces are permitted immediately before or after the equal sign in option parameter.
  - -I*path* This option adds *path* to the include-file path list; for example,

    `-I/APW/LIBRARIES/CINCLUDE`
  - -P This option causes progress information (include-file names, function names, and sizes) and summary information (number of errors and warnings, code size, global data size, compilation time, and compilation memory requirements) to diagnostic output.
  - -S *name* This option sets the load segment name for all object segments created by the CMPL command. The linker will assign all these object segments to the same load segment. This option can be overwridden by a `#segment` directive in the source code
  - -U *name* This option undefines the predefined proprocessor symbol *name*. This is the same as writing

    `#undef` *name*

    at the beginning of the source file.

The space between the option and its parameters is optional for example, `-SMUMBLE` and `-S MUMBLE` are equivalent.

- *language2*=(*option* ...) ... This parameter, like the CC=(*option* ...) parameter, allows you to pass parameters directly to specific APW compilers or assemblers. For each compiler or assembler for which you want to specify options, type the name of the language (exactly as shown by the SHOW LANGUAGES command), an equal sign (=), and the string of options enclosed in parentheses. The contents and syntax of the options string is specified in the compiler or assembler reference manual. Note that the APW Shell does no error checking on this string before passing it through to the compiler or assembler. You can include option strings in the command line for as many languages as you wish; if a language compiler is not called, the string for that language is ignored.

  ❖ *Note:* No spaces are permitted immediately before or after the equal sign in this parameter.

Press Apple-Period to stop the compilation after it has begun. The compiler may respond by halting execution and calling the editor with the first line of your source file at the top of the screen, or it may return you to the shell.

The following command compiles the C source file named MYCFILE.SRC and produces an object file named MYCFILE.ROOT. The C-compiler option that adds a prefix to the include-file path list is passed to the C compiler. If any other files are appended, additional object files named MYCFILE.A, MYCFILE.B, and so on, are produced.

```
COMPILE MYCFILE.SRC KEEP=$ CC=(-I/APW/LIBRARIES/CINCLUDE)
```

❖ *Note:* If you have appended a LinkEd file to the end of your program, the link is controlled by the commands in the LinkEd file. In this case, the standard linker is not called.

For more examples and discussion of the use of COMPILE, and its related command CMPL (alias ASML) command, see Chapter 3 of this manual and "Compiling (or Assembling) and Linking a Program" in Chapter 2 of the *APW Reference*.

EDIT

EDIT *filename*

The EDIT command does one of two things. If a file named *filename* already exists, the command EDIT *filename* calls the editor and opens the file *filename*. The editor uses the language the file is already in. If a file named *filename* does not already exist, the command EDIT *filename* calls the editor and a new file called *filename*. The editor uses the default language (CC, ASM65816, or whatever) established by the last language command or the last file edited.

LINK

LINK [+L|-L] [+S|-S] [+W|-W] *file1* [*file2*] [...][KEEP=*outfile*]

The LINK command calls the APW Linker, which links object files to create a load file. You can use this command to link object files created by APW assemblers or compilers, and to cause the linker to search library files. If any unresolved references remain after all of the specified object files and library files have been specified, the library files in prefix 2/ are searched in the order in which they appear in the directory.

The linker is described in detail in Chapter 5 of the *APW Reference*.

- +L|-L If you specify +L, the linker generates a listing (called a *link map*) of the segments in the object file, including the starting address, the length in bytes (hexadecimal) of each segment, and the segment type. If you specify -L, the link map is not produced.
- +S|-S If you specify +S, the linker produces an alphabetical listing of all global references in the object file (called a *symbol table*). If you specify -S, the symbol table is not produced.
- +W|-W If you select +W, the linker stops and waits for a key press when a nonfatal error occurs, to give you the opportunity to read the error message and to decide whether to continue the link. Press Apple-Period to halt execution, or press any character key or the space bar to continue. If you omit this option or select -W, execution continues without pausing when a nonfatal error occurs. Execution terminates immediately when a fatal error occurs, regardless of the setting of this option.
- *file1 file2* ... The full pathnames or partial pathnames, without filename extensions, of all object files to be included, plus the full or partial pathnames of any library files you want to search. Separate the filenames with spaces. The first file you list, *file1*, must have a .ROOT file; for the other object files, either a .ROOT file or a .A file must be present. For example, the program TEST might consist of object files named TEST1.ROOT, TEST1.A, TEST1.B, TEST2.A, and TEST2.B, all in directory /APW/MYPROG/. In this case, you would use /APW/MYPROG/TEST1 /APW/MYPROG/TEST2 for *objectfile*.

  You can also specify one or more library files (ProDOS 16 file type $B2) to be searched. Any library files specified are searched in the order listed. If a library file is listed before an object file, the library file is searched before that object file is linked. Only the segments needed to resolve references that haven't already been resolved are extracted from library files. See the discussion of the MAKELIB command in Chapter 3 of the *APW Reference* for more information on library files.
- KEEP=*outfile* Use this parameter to specify the pathname or partial pathname of the executable load file.

  You can specify a default load filename by using the LinkName shell variable. Shell variables are described in "Variables" later in this chapter. If you do not specify either the KEEP parameter or a LinkName variable, the link is performed but the load file is not saved.

Important

If you do not include any parameters after the LINK command, you are prompted for an input filename, as APW prompts you for any required parameters. Since the output pathname is not a required parameter, however, you are not prompted for it. Consequently, the link is performed, but the load file is not saved unless you have specified a LinkName variable. Note that you can include the KEEP parameter after the pathname you enter in response to the File name prompt.

As an example of the use of the LINK command, suppose you want to link TEST1, consisting of object files TEST1.ROOT, TEST1.A, and TEST1.B. The following command creates the load file MYTEST; no link map or symbol table is produced:

```
LINK 2/START TEST1 KEEP=MYTEST
```

Suppose you want to link TEST1 consisting of object files TEST.1.ROOT, TEST.1.A, and TEST.1.B, search the library file MYLIB, and link TEST.2 consisting of object files TEST.2.A and TEST.2.B. The following command creates the load file MYTEST, printing the link map but suppressing the symbol table. Note that the library file MYLIB is searched before TEST.2 is linked:

```
LINK +L -S 2/START TEST.1 MYLIB TEST.2 KEEP=MYTEST
```

To automatically link a program after assembling or compiling it, use one of the following commands instead of the LINK command: ASML, ASMLG, CMPL, CMPLG, RUN.

If you need to take advantage of the advanced link capabilites provided by the APW Linker, create a file of LinkEd commands and process it using the ALINK command (or by appending it to the last source file when you compile or assemble your program). The advanced linker is described in detail in Chapter 5 of the *APW Reference*.

Important

The LINK command can be used only to process object files and library files; do not try to process a LinkEd file with the LINK command.

❖ *Note:* If you use a COMPILE command followed by a LINK command and if your main entry point is written in C, you must include the pathname 2/START as the first file in the LINK command.

RUN

RUN [*option* ...] *file1* [*file2*] [...] [KEEP=*outfile*]
[NAMES=(*seg1* [*seg2*] [...])] [*language1*=(*option* ...)
[*language2*=(*option* ...)] [...]]

The RUN command compiles (or assembles) one or more source files, links one or more object and library files, and runs the resulting load file. See the description of the CMPL command for a list of options and a description of the parameters. Your compiler or assembler manual describes the default values of the parameters and the language-specific options available.

The RUN and ASMLG commands are aliases for CMPLG.

The RUN command compiles (or assembles), links, and runs a source file or group of files. Its function is identical to that of the CMPLG command. See the description of the CMPL command for a description of the parameters.

Examples of these commands

The following command compiles a source file named MYFILE and writes the object file to disk as the file MYPROG.ROOT:

```
COMPILE MYFILE KEEP=MYPROG
```

The following command compiles the source file named MYCFILE.

```
COMPILE MYCFILE KEEP=MYPROG CC=(-Ddebug -I/APW/MYINCLUDES)
```

Because MYCFILE is a C program, two C-compiler options are passed to the C compiler: the -Ddebug option defines a compiler flag that you can use to conditionally compile debugging code; and the -I/APW/MYINCLUDES option tells the compiler where to search for additional include files.

Appending files

When APW sees a #append directive in a file, it checks the language type of the appended file: if it is not CC, the compiler returns control to the shell, which brings in the appropriate compiler or assembler to open the file. If the appended file is in the same language, the effect is the same as if the files had been concatenated into one file. If they are in different languages, APW begins a new assembly or compilation. This process has curious effects, as you'll see.

Imagine that there are three files, two in C and one in assembly language, each appended to the preceding file:

```
c1
c2
asm1
```

When you use the COMPILE command, c1 and c2 will be compiled together, and then asm1 will be assembled. All symbols in c1 will be available while c2 is being compiled.

Something different happens when you compile the same files appended in a different order:

```
c1
asm1
c2
```

When you use the COMPILE command, c1 is compiled, then asm1 is assembled, and then the C compiler is called afresh to compile c2. Since the compilations were separate, the compiler knows nothing about symbols in c1 when compiling c2.

Partial compilation or assembly

You can sometimes speed up program development by compiling or assembling only the part of a program that you have changed most recently. The APW Assembler has an option NAMES (for the ASSEMBLE, ASML, ASMLG, COMPILE, CMPL, CMPLG, and RUN commands) that lets you perform partial assemblies, and future APW compilers may also support this option. APW C does not support partial compilation. The APW C Compiler will execute a COMPILE command with the NAMES option, but it will compile the entire source file, as if you had omitted the NAMES option.

The linker

The linker takes object files and file segments created by the C compiler and generates load files. The linker resolves external references and creates relocation dictionaries, which allow the system loader to relocate code at load time. The linker supports **data, code,** dynamic and static segments, and library files.

Normally, the linker is called by the shell command LINK, which provides a limited number of options. Additionally, you can control all functions of the linker by using a language-like set of commands called **LinkEd.** LinkEd is for advanced programmers who require maximum flexibility from the system; for most purposes, the ordinary Link commands are adequate. LinkEd commands are described in Chapter 5 of the *APW Reference;* other APW commands are in Chapter 3 of that book.

When you use CMPL to compile and link a series of files in different languages, the last file in the append sequence must be a C file. The files under the library prefix (prefix 2) are searched for unresolved references.

To link manually and search all libraries, use this command:

```
LINK 2/START objectfilename KEEP=loadfilename
```

The *objectfilename* parameters do not have .ROOT extensions. For example, the command

```
LINK 2/START FILE1 FILE2 FILE KEEP=LOADNAME
```

links the files FILE1.ROOT, FILE2.ROOT, and FILE3.ROOT with the file 2/START.ROOT.

The linker searches every library file (of filetype LIB) in the library prefix (2/).

Making a library

The MAKELIB utility allows you to make a library file. Libraries are useful for storing often-used code, because the linker can search a library much faster that an ordinary object file. Chapter 3 of the APW Reference explains how to use MAKELIB.

Files for compiling and linking

To create a program from source files, the compiler usually needs include files and the linker usually needs libraries. Include files, or header files, must be named in #include statements in the source files. Library files are either searched implicitly or can be named in LINK statements or in LinkEd files.

Include-file search rules

Appendix B, "Files Supplied with APW C," contains a list of include files to be used with APW C. If the include-file name is a full pathname, the compiler uses that name. A full pathname begins with a slash (/) and contains at least one embedded slash. A partial pathname does not begin with a slash. (For more information about pathname syntax, refer to the *Apple IIGS Programmer's Workshop Reference* and the *Apple IIGS ProDOS 16 Reference*.)

If the include-file name is a partial pathname, the compiler searches for include files using the rules shown in Table 2-1. The first file successfully opened using these rules is included.

Table 2-1
Include-file search rules

Include-file name	Example	Search for partial pathname
In double quotes	`"CONSTANTS.H"`	Look in the following directories: 1. The directory of the source file that contains the include statement. 2. The current prefix (`0/`) at the time the compiler was invoked. 3. The directories specified by the `-I` option, in the order given. 4. `2/CINCLUDE`
In angle brackets	`<CTYPE.H>`	Look in the directories just described in item 3, and then the directories described in item 4.

Note that ProDOS filenames are not case-sensitive. By convention, filenames and pathnames are notated in uppercase.

Library files

Appendix B, "Files Supplied with APW C," contains a list of library files to be used with C. (If you use the `CMPL` or `CMPLG` command, the files under the library prefix are searched and you can't specify any others). For more information about linking C programs, refer to Chapter 5 of the APW Reference.

You can control which library files are to be searched by using a LinkEd script. If you specify library files, you will usually want to specify the following:

- all Standard C Library files listed in Appendix B
- only the particular Toolbox files you refer to in your program

Chapter 3

Sample Programs

This chapter provides a tutorial example that illustrates the creation of an application in the APW environment. The program includes a `main` routine in C and a subroutine in assembly language. You are shown how to use the APW Editor to create source files in both languages, and how to compile, assemble, link, and run the program.

The purpose of this chapter is to give you a tutorial introduction to compiling and linking a simple multilanguage program in the APW environment. This example is placed in the *APW C Reference,* rather than in the *APW Reference,* because both APW and APW C are needed to run the example, and only owners of APW C can be assumed to have both.

❖ *Note:* The instructions in this chapter assume that you have both the APW Assembler and the APW C Compiler installed in your system. Assembly language is included on your APW disks; the C compiler is on the APW C disk. See Chapter 2 for instructions on installing APW and APW C in your system.

If you have a hard disk, the instructions in this chapter are straightforward. If you have two 3.5-inch drives, you may have to do some disk swapping and tweaking of prefixes to follow these instructions.

This chapter also provides instructions for building a sample desk accessory.

General procedure

This section describes the general procedure that this chapter follows.

❖ *Note:* For simplicity, the words *compiler* and *compile* are used in this chapter to include *assembler* and *assemble.*

1. Set the system language to the language type of the source code you intend to write, open a file for editing, and write the source code for the first part of your program. Save the file to disk.
2. Execute the shell `COMPILE` (or `ASSEMBLE`) command.

   You now have several files on disk: the source-code file and one or more object-code files (the root file and files with alphabetic extensions such as `.A`).
4. Write the next part of the program. This part need not be in the same programming language as the first part. Give this part a different source filename than the first part and a different `KEEP` filename.
5. Execute the shell `COMPILE` command. Debug the program and recompile as necessary until successful.
6. Repeat steps 4 and 5 for each part of the program, until you are sure that each part compiles successfully.
7. Execute the `LINK` command, specifying the root filenames of all of the object files in the program.
8. If you wish, execute the `COMPACT` command to create a more compact version of the load file.

If you prefer, you can write the entire program, including parts in several languages, and compile and link them all at once. Use the CMPL command to compile and link the program. Each source file except the last can end in an #append directive (or the equivalent), or you can specify multiple source files in the CMPL command. Every time an APW compiler executes an #append directive, it checks the APW language type of the file being appended. If the language doesn't match that of the compiler, then the compiler returns control to the shell, which calls the appropriate compiler to continue processing the program. If all compiles are successful, the APW Linker is called automatically. The linker processes the file, writes out any errors, and (if the link was successful), writes the load file to disk.

Writing and editing the sample source code

The sample program shown in this section takes input from the keyboard, converts every letter to uppercase, and prints the result to the screen. It is written with a main segment in C and a subroutine in assembly language. The C routine handles the input and output. The assembly-language routine does the conversion from lowercase to uppercase.

Use the following steps to write the source code for the C routine shown in Figures 3-1 and 3-2. (If you don't feel like typing, look in the directory /APWC/SAMPLES/UPSTR/.)

1. Boot APW and type the following command to set the system default language (the current language) to C. To execute an APW command, press the Return key.

```
CC
```

2. Call the editor to open a file called SAMPLEC with the following command:

```
EDIT SAMPLEC
```

3. Type the following program. Use the cursor keys to move around in the file. The Delete key deletes the character to the left of the cursor. The Tab key moves the cursor for indenting subroutines. Other basic editor commands are given in Table 2-4 of the *APW Reference*.

```
/* Convert all characters taken from standard input to uppercase */
/*  and write the result to standard output.                     */
/* ...                                                           */
/* NOTE: Control-@ terminates the input                          */

#include <stdio.h>
#define MAXLEN 1024

extern void UPSTR();
char *gets();

main(argc, argv)
int argc;
char *argv[];
{
        char str[MAXLEN];
        while  (gets(str) != NULL) {
                        UPSTR(str);
                        printf("%s\n", str);
        }
        return 0;
}
```

4. Press Apple-Q to quit the editor. Press S to save the file to disk, and then press E to exit the editor and return to the shell.

5. Type the following command to set the current language to 65816 assembler.

```
ASM65816
```

6. Call the editor to open file called SAMPLEA with the following command:

```
EDIT SAMPLEA
```

7. Type the following program. Note that the default tab stops are different for assembly language than for C. You must be careful to start the comments past column 40, or you will get a syntax error in line 2.

```
        LONGA   ON                      set Assembler to 16-bit accumulator mode
UPSTR   START                           start of object segment
        LDA     4,S                     get string address (lower 2 bytes)
        STA     $AA                     store into direct page
        LDA     6,S                     get string address (bank byte+extra byte)
        STA     $AC                     store into direct page
LOOP    SEP     #$20                    set processor to 8-bit accumulator mode
        LONGA   OFF                     set Assembler to 8-bit accumulator mode
        LDA     [$AA]                   get next byte in string
        BEQ     FINISH                  if 0, end of string
        CMP     #$61                    is character < 'a' ($61) ?
        BCC     ITERATE                 if so, go to next character
        CMP     #$7B                    is character >'z' ($7A) ?
        BCS     ITERATE                 if so, go to next character
        SEC                             convert the character to uppercase by
        SBC     #$20                       subtracting $20
        STA     [$AA]                   store character back in string
ITERATE REP     #$20                    set processor to 16-bit accumulator mode
        LONGA   ON                      set Assembler to 16-bit accumulator mode
        CLC                             increment string address by 1
        LDA     $AA
        ADC     #1
        STA     $AA
        BCC     LOOP                    if carry gets set, you just crossed bank
        INC     $AC                        boundaries so you increment bank
        BRA     LOOP                    get next character
FINISH  REP     #$20                    set processor to 16-bit accumulator mode
        LONGA   ON                      set Assembler to 16-bit accumulator mode
        RTL                             return to C routine
        END                             end of object segment
```

8. Press Apple-Q to quit the editor. Press S to save the file to disk, and then press E to exit the editor and return to the shell.

Creating object code: compiling and assembling

To compile and assemble your programs, use the following commands:

```
COMPILE SAMPLEC  KEEP=SAMPLEC.O
ASSEMBLE SAMPLEA  KEEP=SAMPLEA.O
```

❖ *Note:* If you have two 3.5-inch drives and no hard disk, you will have to compile using the APWC disk and assemble using the APW Assembler disk. Use this series of commands:

```
COMPILE SAMPLEC  KEEP=SAMPLEC.O
UMC
ASSEMBLE SAMPLEA  KEEP=SAMPLEA.O
MC
```

If an APW compiler finds a fatal error (one that prevents the compilation from continuing), it writes out an error message to standard output (normally the screen), and passes control to the APW Editor, which loads the source file that the compiler was working on, placing the line that caused the error in the middle of the screen.

If your first attempt was not successful, correct the source code and try again. Repeat this process until the module compiles successfully. Remember to save the source file each time you make changes: the disk file is updated only when you save it.

The following files should be on your disk after using these commands:

SAMPLEC	C source code
SAMPLEA	65816 source code
SAMPLEC.O.ROOT	object segment created by the C compiler
SAMPLEA.O.ROOT	object segment created by the assembler

Alternatively, you can compile both files in one operation, if you are using a hard disk. To do this, you can add a line to the file SAMPLEC as follows:

1. Reopen the file in the editor with the following command:

```
EDIT SAMPLEC
```

2. Press Apple-9 to jump to the end of the file. Add the following line to the file:

```
#append "SAMPLEA"
```

3. Press Apple-Q to quit the editor, S to save the file, and E to exit the editor.
4. Now when you use the following command, the shell calls the C compiler to compile the C routine and then calls the APW Assembler to assemble the 65816 routine:

```
COMPILE SAMPLEC KEEP=SAMPLE.O
```

The following files should be on your disk after using this command:

SAMPLEC	C source code
SAMPLEA	65816 source code
SAMPLE.O.ROOT	first object segment created by the C compiler
SAMPLE.O.A	object segments created by the assembler

Creating load files: linking

When you execute the LINK command, the APW Linker combines all object segments that have the same load-segment name into the same load segment, and places the entire program into a single load file with the KEEP filename you specified. (For a discussion of object segments and load segments, see "APW C Concepts" in Chapter 1.)

Important

Be sure to include the KEEP parameter in the LINK command. If you do not specify a KEEP filename in the LINK command, no load file is saved to disk.

There are two ways to link the object files you have just created. In the first way, if you did *not* add the #append directive to the end of the C routine, use the following command to link the object files into a single executable load file:

```
LINK 2/START SAMPLEC.O SAMPLEA.O KEEP=SAMPLE
```

The first file listed links the file START.ROOT in the library prefix. This file must be linked to the beginning of every program when the main segment is in C.

The load file is named SAMPLE.

The following files should be on your disk after using this command:

SAMPLEC	C source code
SAMPLEA	65816 source code
SAMPLEC.O.ROOT	object segment created by the C compiler
SAMPLEA.O.ROOT	object segment created by the assembler
SAMPLE	load file

In the second way, if you *did* add the #append command to the end of the C routine, use the following command to link the object files into a single executable load file:

```
LINK 2/START SAMPLEC.O KEEP=SAMPLE
```

The following files should be on your disk after using this command:

SAMPLEC	C source code
SAMPLEA	65816 source code
SAMPLE.O.ROOT	first object segment created by the C compiler
SAMPLE.O.A	object segments created by the assembler
SAMPLE	load file

Running your program

To run the program you just created, use the following command:

```
SAMPLE
```

Each character you type is printed on the screen as you type it. Press Return to have the program retype the line in all uppercase. Press Control-C to terminate the program. The following sequence illustrates the use of this routine. The characters in boldface are the ones you type (remember to press Return at the end of each line you type):

```
#SAMPLE
Now is the Time for aLL good PeoPle to Buy an Apple IIgs
NOW IS THE TIME FOR ALL GOOD PEOPLE TO BUY AN APPLE IIGS
Granny Smith is always getting her apples into a jam
GRANNY SMITH IS ALWAYS GETTING HER APPLES INTO A JAM
Control-@
#
```

With this routine, you can use I/O redirection to convert the characters in a file to uppercase. The following command converts all characters in the file `TEXT.IN` to uppercase and writes them out to the file `TEXT.OUT`:

```
SAMPLE  <TEXT.IN  >TEXT.OUT
```

The file `TEXT.OUT` contains the output that would have appeared on the screen; that is, each line of text in the file `TEXT.IN` is printed, followed by the same line converted to uppercase.

Creating a compact load file

As a final step in program development, you can run the Compact utility program. Compact converts a load file to the most compact form provided by the object module format. If your load file is named `SAMPLE`, type the following line and press Return:

```
COMPACT SAMPLE -O SAMPLE.CMPCT -R
```

Compacted load files take up less space on disk and load faster than noncompacted load files. The `SAMPLE` program you created here, for example, should be about 31 blocks in size (as shown in a catalog listing), while `SAMPLE.CMPCT` should be about 25 blocks.

The Compact utility writes to the screen an account of the records it has converted. If you are interested in understanding the format and use of these records, see "Segment Body" in Chapter 8 of the *APW Reference*.

However, not all load files are significantly improved by compacting, so you may want to test both compacted and noncompacted versions of your program before releasing it.

Important

In order to load a compacted load file, you must have version 1.2 or later of the System Loader on your boot disk.

Building a larger application: BONES

The APW C disk contains a sample application named BONES, which does all the things an application needs to do. It is located in the directory /APWC/SAMPLES/BONES/. It comprises the following files:

MAKE	Build EXEC file
BONES.CC	Implements most of BONES
INIT.CC	Initializes tools
DATA.ASM	Data structures for windows and menus
STACKMIN.ASM	Allocates stack for BONES

To build the application, set the prefix 0/ to the directory /APWC/SAMPLES/BONES/ and type

```
MAKE
```

To run the application, type

```
BONES
```

or launch it from the Program Launcher.

Writing desk accssories in APW C

A desk accessory is a small program that a user can run without shutting down an already-running application. The Apple IIGS supports two different kinds of desk accessories:

- **Classic desk accessories** (CDAs) run in a nondesktop environment. The CDA interrupts the application and gets full control of the computer. An example of a CDA is the Control Panel. The APW C Compiler does not support classic desk accessories.
- **New desk accessories** (NDAs) run in a desktop environment: they operate in a window and are subject to the same rules that govern event-driven applications. They are not stand-alone applications, however, because they rely on another application to start up the Apple IIGS tools.

Neither type of desk accessory has much extra programming overhead apart from the actual task it performs. Both types depend heavily for support upon the Desk Manager tool set.

Writing new desk accessories in APW C

All new desk accessories are loaded from the disk at boot time. When an NDA gets control from the Desk Manager, the processor is in full native mode. By convention, the NDA can assume that the tools shown in Table 3-1 have already been loaded and started up. If the NDA needs any other tool sets, it must load and start them up itself.

Table 3-1
Tool sets loaded and available to new desk accessories

Tool set
QuickDraw II
Event Manager
Window Manager
Control Manager
Dialog Manager
Menu Manager
Line Edit
Scrap Manager

The NDA may also assume that the Print Manager is available, although it is not necessarily loaded and started up.

Important

If one of these tool sets has not been loaded and the NDA needs it, the NDA should issue an error message.

An NDA has a structure fundamentally different from that of a desktop application. One difference is that it has no event loop—it relies on the application's event loop and the Desk Manager to open it, probe into action, and close it. Another difference is that it consists of only four routines:

- The Desk Manager calls the *init* routine to initialize the NDA when the Desk Manager starts up, and again when it shuts down.
- The Desk Manager calls the *open* routine when the NDA is selected by the user from the Apple menu. The open routine opens the desk accessory window and returns a pointer to it.
- The Desk Manager calls the *action* routine in response to an event within the NDA window, or when a specified time period has passed, or if a selection has been made from an NDA menu or the Edit menu, or in other special cases. The action routine performs whatever tasks the NDA was designed for. An *action code* passed in the accumulator tells the NDA why it was called.
- The Desk Manager calls the *close* routine to close the desk accessory window.

The processor is in full native mode on entry into all four routines.

The basic procedure followed by each of the four NDA routines is as follows:

1. Call `SaveDB`. (Needed only if you reference variables in `~globals`.)
2. Save important global variables, such as the application's current `GrafPort`.
3. Save the work area pointers of any tools you need to use that are not in Table 3-1.
4. Initialize the tools you need to use.
5. Depending on the action code received, take appropriate action. A desk accessory must use the stack or request needed space from the Memory Manager.
6. Restore the work area pointers that you modified in step 3.
7. Restore the global values and return to the Desk Manager.
8. Call `RestoreDB`. (Needed only if you reference variables in `~globals`.)

You must start the NDA with an identification section that specifies the pointers to the four routines, the NDA's period (how often it runs), and its menu line (text defining its title on the Apple menu). For example, the identification section could look like this:

"NDA's have the ProDOS file type 0xB8. On disk, they must reside in the */volumename/*SYSTEM/DESK.ACCS/ subdirectory."

A sample C desk accessory

A sample NDA, written in C, is in the directory /APWC/SAMPLES/DA/. It comprises the following files:

IDLEHEADER.ASM	NDA identification section with pointers to four routines
CIDLE.C	Implements *init, open,* and *close* (steps 1-4 and 6-8, just given)
USERIDLE.C	Implements *action* (step 5): change this to create your own NDA
DB.ASM	Implements SaveDB and RestoreDB
MAKE	Build EXEC file

To build the desk accessory, set the prefix 0/ to the directory /APWC/SAMPLES/DA/ and type

```
MAKE
```

To run the desk accessory, copy it into the directory */volumename/*SYSTEM/DESK.ACCS/ on a disk named *volumename* and boot that disk.

Part II

Language Reference

Chapter 4

The APW C Language

The information provided in this chapter supplements *The C Programming Language* by Kernighan and Ritchie. Where the K and R language definition leaves choices to the implementers, this chapter describes how these aspects of C have been implemented on the Apple IIGS. Where Apple has modified or extended the K and R language definition, this chapter documents the changes.

Language definition

This section describes the APW C language, including language extensions such as type `void`, type `enum`, and the SANE data types, and how to call Pascal-style functions. It also describes APW C's in-line assembler.

Variable names

The compiler limits the length of each local variable name to 1000 characters. Global variable names and function names are limited to 250 characters by the object-module format. Therefore, different function names whose first 250 characters are identical will be treated as different functions by the compiler, but will be treated as the same function by the linker.

Data types

Table 4-1 lists the arithmetic and pointer types available in APW C and shows the number of bits allocated for variables of these types. Types `short` and `long` represent 16-bit and 32-bit integers, respectively. The machine type `int`, a 16-bit integer on the Apple IIGS, is the type the 65C816 uses most efficiently. Pointers require 32 bits. Enumeration types require 16 bits. Types `short`, `int`, and `long` use two's-complement representation. Type `char` is unsigned. Note that the Apple IIGS has no signed 8-bit type: `char` and `unsigned char` are identical. Naturally, a prudent programmer will make no assumptions about features not guaranteed to be portable.

Table 4-1
Size and range of data types

Data type	Bits	Description
`char`	8	Range 0 to 255
`unsigned char`	8	Range 0 to 255
`short`	16	Range –32,768 to 32,767
`unsigned short`	16	Range 0 to 65,535
`int`	16	Range –32,768 to 32,767
`unsigned int`	16	Range 0 to 65,535
`long`	32	Range –2,147,483,648 to 2,147,483,647
`unsigned long`	32	Range 0 to 4,294,967,295
`enum`	16	Range 0 to 65,535
`*`	32	Pointer types
`float`	32	IEEE single-precision floating point
`double`	64	IEEE double-precision floating point
`comp`	64	SANE signed integral values
`extended`	80	IEEE extended-precision floating point

❖ *Note:* Some programs assume that `sizeof(int) = sizeof(char *)` may not work properly under APW C because an `int` is 2 bytes long and a pointer is 4 bytes.

You can find more information about types in Table 4-2, given later in this chapter.

Numeric constants

Integer constants in the range of `long` are treated as type `long`. Integer constants in the range of `unsigned long` are treated as type `long` unless you explicitly include a cast to type `unsigned long`.

This point is important for those few cases where the `long` constant has the most significant bit set, because the compiler may seem confused about whether such constants have large positive values (which are stored as 32 bits with the most significant bit set) or negative values.

For instance:

```
(4000000000 < 0)
```

is true, but

```
((unsigned long)4000000000 < 0)
```

is false.

Integer constants outside the union of the ranges of the types `long` and `unsigned long` are treated as type `extended`. For example, the initialization statement

```
long i = 6000000000;
```

is incorrect because 6,000,000,000—being too big for the `long` type—is interpreted as an `extended` value. However, the initialization statement

```
unsigned long i = 4000000000;
```

is correct because 4,000,000,000 is within the range of `unsigned long` values.

Type void

The `void` keyword tells the compiler that the function being declared does not return a value. Calls to functions of type `void` may not be used in expressions, where a value is required. (See "Pascal-Style Functions" later in this chapter.)

Type enum

Type enum is a type analogous to the enumeration types of Pascal. Its syntax is similar to that of the `struct` and `union` declarations:

enum-specifier:
`enum` { *enum-list* }
`enum` *enumeration-tag* { *enum-list* }
`enum` *enumeration-tag*

enumeration-tag:
identifier

enum-list:
enumeration-declaration
enumeration-declaration , *enum-list*

enumeration-declaration:
identifier
identifier = *constant-expression*

Like the structure tag in a *struct-specifier* parameter, the optional *enumeration-tag* in *enum-specifier* names a particular enumeration type and allows you to define other objects of that type. For example,

```
enum color {chartreuse, burgundy, claret, winedark};
. . .
enum color *cp, col;
```

makes `color` the *enumeration-tag* of a type describing various colors and then declares `cp` as a pointer to an object of that type and `col` as an object of that type. The identifiers in *enum-list* are declared as constants and may appear wherever constants are required.

If no enumerators with a *constant-expression* appear, the value of each constants begins at 0 and increases by 1 as the declaration is read from left to right. Each enumerator with a *constant-expression* is given the value indicated. Each enumerator without a *constant-expression* is given a value one greater than the enumerator before it. This means that two or more enumerators with *constant-expressions* can be assigned the same constant value, and that an enumerator without a *constant-expression* may have the same value assigned by the compiler as another enumerator with a *constant-expression* in the same enumeration list. Consider some examples:

```
enum digit {zero,one,two,three,four,five,six,seven,eight,nine } num;
```

has the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9;

```
enum mixedup {a,b,c,d = 1,e,f } mix;
```

has the values 0, 1, 2, 1, 2, 3;

```
enum zapped {g = 1, h,i,j =2,k,l} zap;
```

has the values 1, 2, 3, 2, 3, 4; and

```
enum ok {m=45,n,o,p=100,q,r};
```

has the values 45, 46, 47, 100, 101, 102.

If you declare values, it is safest to declare all of them.

Each *enumeration-tag* and *enumeration-constant* must be unique. Unlike structure tags and members, they are drawn from the set of ordinary identifiers. Objects of a given enumeration type have a type that is distinct from objects of all other types.

Enumeration types are allocated the amount of space required by the smallest predefined type that allows representation of all literal values specified by the enumeration. The predefined types considered are `unsigned char` (8 bits) and `unsigned short` (16 bits).

Register variables

Most versions of C support register variables. The function of register variables is undefined in the Apple IIGS as a result of the small number of registers available on the 65C816 microprocessor. Use of the `register` declaration neither optimizes nor pessimizes your code: the C compiler generates equally efficient code whether or not your source code contains `register` declarations.

Structures

Structures may be assigned, passed as parameters, and returned as function results. The left and right sides of a structure assignment must have identical types. Similiarly, actual and formal parameters must have identical types. Equality comparison for structures is implemented, provided the structures have the same type. (The equality test may give unpredictable results if a structure contains a `union`.)

Because the 65C816 is a byte-oriented machine, data structures can be aligned on byte boundaries. For this reason, APW C does not pad structures to ensure word alignment.

Important

In functions that return structures, if an interrupt occurs during the return sequence and the same function is called reentrantly during the interrupt, the value returned from the first call may be corrupted. This problem can occur only in the presence of interrupts. Recursive calls are quite safe.

Reserved symbols

__LINE__ is a reserved preprocessor symbol whose value is the current line number within the current source file.

__FILE__ is a reserved preprocessor symbol whose value is a character string consisting of the current filename.

__LINE__ and **__FILE__** begin and end with two underscore characters.

The symbol **AppleIIgs** is predefined for use in conditional compilation. It can be used to distinguish C code written for the APW C Compiler from C code written for, say, the MPW C Compiler. The symbol has the value 1, as if a statement of this form had appeared at the beginning of the source code:

```
#define AppleIIgs 1
```

The symbol **APW** is predefined for use in conditional compilation. It can be used to distinguish C code written for the APW C Compiler from C code written for some other compiler. The symbol has the value 1, as if a statement of this form had appeared at the beginning of the source code:

```
#define APW 1
```

The symbol **WD65816** is predefined for use in conditional compilation. It can be used to distinguish C code written to run on the Western Design Center 65SC816 from C code written to run on some other microprocessor—even for some other variation of 65816.The symbol has the value 1, as if a statement of this form had appeared at the beginning of the source code:

```
#define WD65816 1
```

An `ifdef` statement can test the `AppleIIgs`, `APW`, and `WD65816` symbols.

Standard Apple Numeric Environment extensions

APW C has built-in support for SANE. In combination with the SANE routines in `CLIB`, the language composes a scrupulously conforming extended-precision implementation of the IEEE Standard for Binary Floating-Point Arithmetic (754). SANE provides an extra data type for storing large integral values and basic functions for application development. APW C recognizes the SANE data types, uses SANE for all C floating-point operations and conversions, and correctly handles NaNs (Not-a-Number) and infinities in comparisons and in ASCII–binary conversions. Furthermore, source programs from other C implementations—if they are written using only `float` and `double` type, and standard C operations—will compile and run under APW C without modification.

Much of SANE is provided through the run-time library `CLIB` and the include file `SANE.H`. However, to use extended-precision arithmetic efficiently and effectively, and to handle IEEE NaNs and infinities, some extensions to standard C are required, including use of the `extended` data type.

A change from `double` to `extended` as the basic floating-point type is the most important difference from standard C. Because C was originally developed on the DEC PDP-11, the PDP-11 architecture is reflected in standard C in the use of `float` and `double` as floating-point types, with `double` being the basic type. Thus, floating-point expressions are evaluated to `double`, anonymous variables are `double`, and floating-point parameters and function results are passed as `double` values. However, the low-level SANE arithmetic (as well as the Intel 8087, Motorola 68881, and Zilog Z8070 floating-point chips) evaluates arithmetic operations to the range and precision of an 80-bit `extended` type. Thus, `extended` naturally replaces PDP-11 `double` as the basic arithmetic type for computing purposes. The types `float` (IEEE single), `double`, and `comp` serve as space-saving storage types, just as `float` does in standard C. The `comp` type, which is a 64-bit type for storing integral values, is a SANE extension. It has two properties that suit it to accounting applications: it is sufficiently large to represent the U.S. national debt in Argentine pesos, and it has a NaN value to record overflows and other exceptions.

The IEEE Standard specifies two kinds of special representations for its floating-point formats: NaNs and infinities. APW C expands the syntax for I/O to accommodate NaNs and infinities, and includes the treatment of NaNs in relationals as required by the IEEE Standard.

The SANE extensions to standard C are *backward-compatible*: programs written with only the `float` and `double` floating-point types and standard C operations compile and run without modification. All intermediate values are computed in the `extended` type, an 80-bit floating-point type, and the results are returned to the types specified in the program. SANE does not affect integer arithmetic.

The *Apple Numerics Manual* contains detailed documentation of SANE. The *Apple IIGS Toolbox Reference* contains detailed documentation of the Apple IIGS SANE Toolset, which makes SANE available on the Apple IIGS.

Constants

Numeric constants that include floating-point syntax—a point (.) or an exponent field—or that lie outside the union of the ranges of the `long` and `unsigned long` types are of type `extended`. Binary-to-decimal conversion of constants is performed at compile time (and hence is governed by the default numeric environment: see the section "Numeric Environment" later in this chapter).

Expressions

The SANE types—`float`, `double`, `comp`, and `extended`—can be mixed in expressions with each other and with integer types in the same manner that `float` and `double` can be mixed in standard C. An expression consisting solely of a SANE-type variable, constant, or function is of type `extended`. An expression formed by subexpressions and an arithmetic operation is of type `extended` if either of its subexpressions is. Expressions of type `extended` are evaluated using extended-precision SANE arithmetic, with conversions to type `extended` generated automatically as needed. Parentheses in `extended`-type expressions are honored: the compiler will not rearrange terms in violation of parentheses. Initialization of external and static variables, which may include expression evaluation, is performed at compile time. All other evaluation of `extended`-type expressions is performed at run time.

Comparison involving a NaN

The result of a comparison involving a NaN operand is **unordered.** The usual set of comparison results—less than (<), greater than (>), and equal to (==)—is expanded to include unordered. For example, the negation of "*a* is less than *b*" is not "*a* is greater than or equal to *b*" but "*a* is greater than or equal to *b*, or *a* and *b* are unordered." The `CLIB` function `relation` tests all four alternatives.

Parameters and function results

A numeric actual parameter passed by value is an expression and, hence, is of `extended` or `integer` type. All `extended`-type arguments are passed as `extended` values. Similarly, all results of functions declared `float`, `double`, `comp`, or `extended` are returned as `extended` values.

Numeric input and output

In addition to the usual syntax accepted for numeric input, the Standard C Library function `scanf` recognizes the string "`INF`" as infinity and the string "`NAN`" as a NaN. "`NAN`" may be followed by parentheses, which may contain an integer (a code indicating the NaN's origin). "`INF`" and "`NAN`" preceded by a sign and are case-insensitive. The `scanf` specifiers for SANE types extend standard C as follows: conversion characters `f`, `e`, and `g` indicate type `float`; `lf`, `le`, and `lg` indicate type `double`; `mf`, `me`, and `mg` indicate type `comp`; and `ne`, `nf`, and `ng` indicate type `extended`.

The Standard C Library function `printf` writes infinities as the string "INF" and NaNs as the string "NAN(*ddd*)", where *ddd* is the NaN code. "INF" and "NAN(*ddd*)" may be preceded by a minus sign.

Numeric environment

The **numeric environment** comprises the rounding direction, rounding precision, halt enables, and exception flags. IEEE Standard default settings—rounding to nearest, rounding to extended precision, and all halts disabled—are in effect for compile-time arithmetic (including decimal-to-binary conversion). Each program begins with these defaults and with all exception flags clear. Functions for managing the environment are included in the library `CLIB`. The compiler, in optimizing, will not change any part of the numeric environment, including the exception-flag setting, which is a side effect of arithmetic operations.

About the SANE routines in CLIB

The SANE routines provide the basic tools for developing a wide range of applications. They include the following:

- logarithmic, exponential, and trigonometric functions
- financial functions
- random-number generation

- conversions between binary and decimal formats
- numeric scanning and formatting
- environment control
- other functions required or recommended by the IEEE Standard

Additional information can be found in the SANE Tool Set chapter of the *Apple IIGS Toolbox Reference.*

Programming with IEEE arithmetic

APW C's automatic use of the `extended` type produces results that are generally better than those of other C systems. For example, extended precision yields more accuracy and extended range, avoiding unnecessary underflow and overflow of intermediate results. You can further exploit the `extended` type by declaring all floating-point temporary variables to be of type `extended`. Doing this is both time-efficient and space-efficient, since it reduces the number of automatic conversions between types. External data should be stored in one of the three smaller SANE types (`float`, `double`, or `comp`), not only for economy but also because the `extended` format may vary between SANE implementations. As a general rule, use `float`, `double`, or `comp` data as program input; `extended` arithmetic for computations; and `float`, `double`, or `comp` data as program output.

In many instances, IEEE arithmetic allows simpler algorithms than were possible without IEEE arithmetic. The default overflowing to infinity enlarges the domain of some formulas. For example, $1+1/x^2$ will be computed correctly even if x^2 overflows. Running with halts disabled (the default), a program will never crash due to a floating-point exception because a suitable default value can be returned instead. Hence, by monitoring exception flags, a program can test for exceptional cases after the fact. The alternative—screening out bad input—is often infeasible and sometimes impossible.

The in-line assembler

The APW C in-line assembler obviates the need for a separate assembler. You can implement general control structure, input/output, and complex data structures in C, while coding certain low-level routines in assembly language within the same module. The problem of interfacing C functions to assembly-language functions and vice-versa is eliminated, because calling sequences can be written in C for functions coded in assembly language. Programs can first be developed in C to debug algorithms and to generate a working prototype quickly. The functions that consume the most time (generally less than 10% of the code) can then be re-coded in assembly language. Because of the efficiency of the APW C code generator, such a hybrid approach yields execution speeds comparable with those of pure assembly-language code, while retaining the ease of modification and maintenance of a pure high-level-language approach.

Use of assembly language decreases readability, exacerbates debugging headaches, and drastically reduces portability, so you must use discretion when considering functions for hand translation. There are some situations where speed is critical, most notably graphics. Such applications frequently involve system or machine dependencies anyway, so portability is not an issue. In such cases, the availability of in-line assembly language is a great benefit.

In-line assembly-code declarations and definitions

Your C program can contain assembly code in line. Anywhere that a statement is legal, you can insert a series of assembly-language statements with this format:

`asm{` *assembly-language-statements*`}`

Anywhere that a function definition is legal, you can have a definition with this format:

`asm(`*external-name*`) {` *assembly-language-statements*`}`

This function can be called in the same way as a C function called *external-name*. Here, *external-name* is the entry point of the segment containing the assembly-language code.

In-line assembler syntax

The assembler syntax is basically the same as that used in the APW Assembler. There are far fewer assembler directives: only `dcb`, `dcw` and `dcl` are supported. Macros are not supported either; however, the compiler's preprocessor is active within in-line assembly.

The general syntax for in-line assembly language follows. Here is the syntax for metasymbols:

Metasymbol form	Meaning
item	*item* is replaced by an actual item.
item ...	*item* may be repeated.
[*item*]	*item* is optional.
choice1 | *choice2*	either *choice1* or *choice2* must appear, but not both.

Here are the syntax rules for statements:

asm-function ::= `asm(` *func-name* `){` *asm-line* ... `}`
asm-statement ::= `asm{` *asm-line* ... `}`
asm-line ::= *label* : | *op-code* [*operand*] [*comment-stuff*]
comment-stuff ::= ; *comment* | /* *comment* */

In-line assembly code may appear anywhere in your program; it is not necessary to place it inside a function. The *asm-function* format is used in this case. An *asm-statement* may appear anywhere that a C statement is legal within a regular C function. C variables may be referred to by name. All `auto` variables and parameters are accessed with direct-page addressing; global and `static` variables are accessed with long absolute addressing.

Opcodes are the same as in the Western Design literature and may be given in uppercase or lowercase. Because expansion of `#define` macros is performed within sections of assembly language, you are free to rename instructions or registers.

Each line of assembly language may consist of one or more instructions, optionally followed by a semicolon and comment text. Comments may also be given as C comments. Note that you can use `#define` statements to create simple macros using the multiple-statement-per-line feature. Within macros, C-style comments must be used instead of the normal semicolon-to-end-of-line assembly-language comments.

An expression giving a displacement value is permitted after an identifier. The expression is a C-style constant expression that is added to or subtracted from the identifier. All constant expressions may use C-style constants (such as `\012` or `0x40`) and may use the constant operators listed in section 15 of Appendix A of Kernighan and Ritchie. Note that `$1234` is invalid syntax for hexadecimal constants: use `0x1234` instead.

In addition, the unary <, unary >, unary ^, and unary | are all recognized by this assembler, and have the same meaning as they do in the APW assembler. In operand expressions, any identifier must be placed first in the expression; in other words, the instruction

```
lda     a+2
```

is legal, but the instruction

```
lda     2+a
```

is not.

The syntax for reserving a byte, word, or long is

```
dcb     expression
dcw     expression
dcl     expression
```

respectively.

An identifier that appears after `dcl`, `dcw` or `dcb` means to emit the address of the identifier. If the identifier is `auto`, the offset into the direct page is emitted; otherwise, its absolute address is emitted. Direct page offsets are modulo 256, so if you have more than 256 bytes of `auto` variables, the compiler may silently generate incorrect offsets.

All labels given default to local-code labels unless you've previously declared them as something else. This means that all functions called, for example, must be declared or defined previously in C. You may only use a *label* as a destination for a branch; you may not read or store values using a label. This restriction exists to ensure that code segments are pure code, which is a requirement of the loader for restartable applications.

```
/*
       An example of a macro to use in assembly language
*/

#define MLI16 0xE100A8
#define PRODOS(n, a) jsl MLI16  dcw n  dcl a
```

You have the ability to obtain offsets and values of structure members, as shown in this example:

```
typedef struct _don {
        int x;
        int y;
        int z;
} don;

don globaldon;

main()
{
don localdon;
        asm {
        lda     #don.y                  ; load offset of structure member
        lda     globaldon.y             ; load global value of structure member
        lda     localdon.z              ; load local value of structure member
        lda     #|(sizeof(int) + (~123) * '\0123')
        }
}
```

Finally, this assembler has a few differences with respect to the APW Assembler. The `jmp` opcode always generates short jumps, for long jumps, you must use the `jml` opcode. Similarly, `jsr` always generates short jumps; for long jumps, you must use `jsl`.

Here are some examples of correct and incorrect syntax:

Correct	Incorrect
`pei dp`	`pei (dp).`
`pea #expr`	`pea expr.`
`mvn #dst, #src`	`mvn src, dst.`
`mvp #dst, #src`	`mvp src, dst.`

Some synonyms for opcodes (such as `swa` for `xba`) are not supported. You can easily work around this by doing, for example,

```
#define tda tdc
#define swa xba
```

The assembler will always generate 16 bits of operand for instructions like

```
lda #0
```

This means that the assembler is meant to generate instructions in full native mode.

Pascal-style functions

The function-calling conventions used by APW C and by conventional Pascal implementations differ in the order of parameters on the stack, the type coercions applied to parameters, and the location of the return result. Like the Macintosh Toolbox, the Apple IIGS Toolbox adheres to Pascal-style calling conventions. APW C has been extended to allow you to use both C-style and Pascal-style calling conventions. The specifier `pascal` in a function declaration or definition indicates a Pascal-style function. This extension is intended to allow for the addition of Pascal and other languages to APW.

Pascal-style function declarations

A function or procedure written using Pascal-style calling conventions can be called from APW C. Before the function or procedure can be called, it must be declared as an external function. Here is the general form for a declaration:

`[extern] pascal` [*result-type*] *func-name* `() ;`

This declaration says that the Pascal procedure named *func-name* can be called from your program, returning a result of type *result-type*.

For example, the `DrawText` procedure would be defined in Pascal as follows:

```
PROCEDURE DrawText (textBuf: Ptr;
     firstByte, byteCount: integer);
```

The syntax for declaring this procedure so that it can be called from APW C is

```
extern pascal void DrawText();
```

To make the code more informative, you can list the parameters in a comment:

```
extern pascal void DrawText();
     /* Ptr textBuf;
     short firstByte, byteCount; */
```

The inline declaration

An `inline` declaration is used for declaring Apple IIGS tool routines. Its syntax is

`[extern] pascal` [*result-type*] *func-name* `() inline(`*m*`,` *n*`);`

This declaration says that the tool routine with tool-call number *m* and Tool-Locator entry-point *n* can be called by the function name *func-name* and that *func-name* returns a result of type *result-type*. The `pascal` keyword is necessary because the tools use Pascal-style conventions. If the tool returns an error, it can be found in `_toolErr`, a global integer variable declared inside `CLIB`. For example, where `m` and `n` are integer constants, the C source code

```
extern pascal void foo() inline(m,n);
main()
{
foo();
}
```

generates code like this:

```
        ...             ; code to set up the stack frame:
        LDX     #m
        JSL     n
        BCS     OVER
        LDA     #0
OVER    STA     >_toolErr       ; _toolErr == 0 iff no error
        ...             ; code to clean up the stack frame
```

❖ *Note:* In APW C, the names of global variables and functions in the object file are identical to their names in the source file. There are no prepended underscores, folding to uppercase, or other perversions of the source names.

Pascal-style function definitions

A C function definition (the actual function), like a function declaration, can also be preceded by the `pascal` specifier. The C compiler then produces code that adheres to Pascal-style calling conventions and the function can be called using these conventions.

The APW syntax for defining this procedure as a C function is

`pascal` [*result-type*] *func-name* (*formal-parameter-list*) { *statement-list*}

For example, the following C function could be called from Pascal:

```
pascal void MyText(byteCount,textAddr,numer,denom)
     short byteCount;
     Ptr textAddr;
     Point numer,demon;
{
     ...
}
```

The corresponding Pascal function declaration would be

```
PROCEDURE MyText(bytecount: INTEGER; textAddr: Ptr;
     numer,denom: Point);
```

For compatibility with Pascal and assembly language, the compiler converts the names of Pascal-compatible functions to uppercase before writing them to the object file. When they are called in C programs, these routines should be capitalized exactly as they were declared in C. Pascal-compatible functions whose names differ only in their capitalization will become duplicate declarations when their names are converted to uppercase by the compiler; therefore, such names should be avoided.

Pascal-style strings: \p

One of the complications of calling Pascal-style functions from C is that the two languages have different conventions for handling strings. A C-style string is a set of characters followed by a null byte; a Pascal-style string is a count byte *n*, followed by a set of *n* characters. Conveniently, these two forms are the same length, so conversion from one to the other is not hard. The functions `c2pstr` and `p2cstr` perform runtime conversions between the two types of strings.

If you wish to call a Pascal-style function that expects a Pascal-style string, you can use the Apple extension to the standard C character escapes: `\p`. When the compiler encounters this escape sequence at the beginning of a string, it substitutes for the `\p` the character value equivalent to the number of non null characters in the remainder of the string. Thus a string constant is created that is equivalent to a Pascal-style string. Since it is also a C-style string, it is also terminated by the null character: this character is not included in the character count.

You can use it like this:

```
WriteString("\pHello, world.\n");
```

Parameter and result data types

C and Pascal support different data types. When writing a Pascal-style function declaration in C, a translation of the parameter types and function-result type (from Pascal to C) is therefore required. Often this translation is obvious, but some cases are surprising.

Table 4-2 summarizes this translation. Find the Pascal parameter or result type in the first column. Use the equivalent C type found in the second column when declaring the function in C. Comments in the table point out unusual cases that may require special attention.

Table 4-2
Parameter and result data types

Pascal Data Type	C Equivalent	Comments
`enumeration`	`enum`	Use identical ordering of the `enumeration` literals.
`var enumeration`	`enum *`	
`enumeration result`	`enum`	
`char`	`char`	Pascal passes `char` parameters as 16-bit values.
`var char`	`char *`	Pascal stores unpacked `char` parameters as 16-bit values.
`char result`	`char`	
`integer`	`int or short`	16-bit signed values
`var integer`	`int * or short *`	
`integer result`	`int or short`	
`longint`	`long`	32-bit signed values
`var longint`	`long *`	
`longint result`	`long`	
`real`	`float *`	Pascal passes `real` parameters as `extended`.
`var real`	`float *`	
`real result`	`float`	
`double`	`double *`	Pascal passes `double` parameters as `extended`.
`var double`	`double *`	
`double result`	`double`	
`comp`	`comp *`	Pascal passes `comp` parameters as `extended`.
`var comp`	`comp *`	
`comp result`	`comp`	
`extended`	`extended *`	
`var extended`	`extended *`	
`extended result`	`extended`	

(continued)

Table 4-2 (continued)
Parameter and result data types

Pascal Data Type	C Equivalent	Comments
`pointer`	`pointer`	32-bit addresses
`var pointer`	`pointer *`	
`pointer result`	`pointer`	
`array`	`array`	Pascal passes `array` parameters by address.
`var array`	`array`	
`array result`	`---`	C does not allow array results.
`record`	`struct`	Pascal passes `record` parameters by value.
`var record`	`struct *`	
`record result`	`struct`	
`set`	`struct`	Pascal passes `set` parameters by value.
`var set`	`struct *`	
`set result`	`struct`	

❖ *Note:* The C `struct` type and the Pascal `record` type do not exactly correspond, because C lacks an equivalent to the Pascal `variant record` type.

Global and external data types

When a C program and a Pascal program use the same global or external variables, they must use types of the same size. This requires care, as you can't be sure whether a given Pascal compiler puts 0..255 into a byte or a word. If possible, use a signed type for a signed type. If you have to pass values from a signed type into an unsigned type or vice versa, you must test the sign bit and perform the appropriate conversions.

How parameters are passed

High-level languages on the Apple IIGS use the stack, and the A and X registers to pass parameters. Assembly-language programs have other means of passing parameters, such as the direct page, but they must use the stack to communicate with C programs, because this is how C expects parameters to be passed. Here's how parameters are passed.

C-style functions

Suppose you declare a typical C-style function:

```
int foo();
```

This function takes three values and returns one result. You can call the function like this:

```
zoo = foo(a,b,c);
```

When the call is executed, the values c, b, and a are pushed, in that order. Function foo returns its result in the A register. The calling program then pulls a, b, and c off the stack and stores the contents of the A register into the variable zoo.

If foo had been 4 bytes long, it would have been returned in the A and X registers, with the high bytes in X and the low bytes in A. Structure and extended results are returned by passing a pointer to them in the A and X registers.

Pascal-style functions

Pascal-style functions use the stack for the return value and also reverse the order of reading parameters. Consider this function:

```
pascal int foobar();
```

This function also takes three values and returns one result. You can call the function like this:

```
x = foobar(a,b,c);
```

When the call is executed, space for the result foobar is pushed onto the stack, and then the values a, b, and c are pushed, in left-to-right order. The routine pulls c, b, and a off the stack, computes foobar, and pushes foobar onto the stack. The calling program then pulls foobar off and copies it to the variable x.

When you write a function, you can declare it as a C-style or a Pascal-style function, thus determining the way the parameters are passed. The C style of passing parameters is more efficient than the Pascal style, but it should be used only with functions that will be called from C and not from Pascal. Whatever language a function is written in—if the function is declared as a Pascal-style function—it can be called from either Pascal or C; if it is declared as a C-style function, it can be called only from C.

Sample program

The following program shows how to use both C-style and Pascal-style parameter-passing conventions. The C main program calls two assembly-language routines, one C-style and one Pascal-style. These routines are declared in-line with asm statements.

```
#include <stdio.h>

/*

   callsamp

   Sample program to illustrate two different function calling
   conventions available from C: "vanilla" C-style parameter
   passing (the default); and Pascal-style parameter passing,
   used most notably to pass parameters to toolbox routines, but
   potentially useful for other things (like calling Pascal functions/
   procedures if in fact the actual Pascal compiler used follows the
   "pascal" parameter-passing conventions.

*/

/* Declare psum as a function of type pascal, so arguments
   will be passed Pascal-style, and return value will be pulled
   from stack.
 */
```

```
pascal int psum();

/* Call csum and psum (sum arguments), and print results. */

main()
{
       printf("c result: %d\n",csum(1,2,3));
       printf("p result: %d\n",psum(1,3,4));
}

/*     Process parameters using C conventions.

       Note that csum and psum are not actually C functions,
       but rather assembly code.

       The assembly routines operate on the stacked parameters
       according to the respective parameter-passing conventions . . .
 */

asm (csum)
{
       lda     4,s     ; Skip return address, get top parameter.
       cmp     #1      ; Should be a 1.
       bne     bad     ; Prove that 1 is pushed last.
       clc
       adc     6,s
       adc     8,s     ; Return with sum in accumulator.
       rtl

bad:
       lda     #0      ; This would only happen if I'm lying,
       rtl             ; so it will never happen.
}

/*     Process parameters using pascal conventions.     */

asm (psum)
{
       lda     8,s     ; Get first pushed parameter.
       cmp     #1      ; Should still be 1.
       bne     bad     ; Prove 1 pushed first.
       clc
       adc     6,s
       adc     4,s
       sta     10,s    ; Save result on stack, above parameters.
                       ; (Result space was pushed onto stack).

       lda     2,s     ; Pull high, bank bytes of return address.
       sta     8,s     ; Put under result.
       lda     1,s     ; Pull low, high of return address.
       sta     7,s     ; Put under bank.
       pla             ; Having shuffled values up,
       pla             ;   pull off storage equivalent to
       pla             ;   size of three parameters pushed
       rtl             ;   return with result on stack.

bad:
       lda     #0      ; This would only happen if I'm lying,
       rtl             ; so it will never happen.
}
```

Implementation notes

A number of details in any language definition are left to the discretion of its individual implementers. Most programs do not rely on these details and, therefore, yield the same results in all implementations. Knowledge of the major differences between implementations can, however, help you avoid reliance on implementation-dependent language semantics. This section explains several areas of the language definition that are specific to APW C.

Size and byte-alignment of variables

Because the 65C816 is a byte-oriented processor, it levies no speed penalty for using odd addresses. Therefore, APW C does not align variables on word boundaries. In particular, enumerated types and structures are not padded to make fields fall on word boundaries.

When you recompile an MPW C program on the APW C Compiler, for example, all padding added by the MPW C Compiler disappears. Any padding you added remains. You can save space and possibly time by removing this padding from data structures and by deleting code that performs word alignment.

Byte ordering

On the 65C816, the microprocessor used in the Apple IIGS, the least-significant byte of a short or long integer has the lowest memory address. This byte ordering is also used on the PDP-11, VAX, 8086, and NS16000 processors. The 68000, IBM/370, and Z8000 processors store the least-significant byte at the highest address. Programs that rely on the order of the bytes within words and long words will not be portable from machines of one of these classes of machines of the other.

Variable allocation

The APW C Compiler allocates static and global variables in the order in which they appear in the source. This is also true for the order of fields within structures.

Variables of type void

The APW C Compiler allows you to declare `void` variables, which take up the same number of bytes as `int` variables, but you can't do anything with them.

Array indexing

Array indexing is done using long arithmetic wherever the compiler cannot determine the actual size of the array (as in `extern int array[];`) or can determine that the size is greater than 64K (0x10000) and, therefore, requires long arithmetic for correct calculation of offsets.

If the compiler determines that the entire array can be accessed using word arithmetic, it may do so, as shown here:

```
extern int array[N]; /* N <= 0x8000 */

char string[] = "It would be hard to create a string long enough to require
                 long indexing, wouldn't it?"

int notTooMany[] = {0,1,2,3,4,5,6,7,8,9};
long larray[0x4000];

long larray[0x4000][0xA];   /* Though the array is too large, the second
                               index will be done with word arithmetic.
                               This is of dubious advantage. */
```

Because word arithmetic is more efficient than long arithmetic, you can use certain tricks to force word arithmetic when speed is important. These tricks apply whenever you only need to access no more than 64K (0x10000) bytes within an array.

1. The form

   ```
   extern int array[0x400];
   ```

   is better than the form

   ```
   extern int array[];
   ```

   (as long as you know how much of the array you need to access).

2. To optimize access to a part of a larger array, place the code in a subroutine and pass a pointer to the first element of the part to the subroutine, as shown here,

   ```
   long array[0x10000] /*This will normally cause long index arithmetic.*/

   main()
   {
   unsigned int i;

            for(i=0; i<4; i++)
                   fill(array+i*0x4000);
   }
   fill(smaller)

   long *smaller;

   {
   unsigned i;

            for (i=0; i < 0x4000; i++)
                   *smaller++ = 0xFFFFFFFF;
   }
   ```

   Calling `fill` four times allows you to fill an array whose actual size in bytes is `0x40000`, using long-arithmetic address calculation only four times, once at each call from `main`. Note that the arithmetic is further optimized by the use of `unsigned` for `i`.

Types unsigned char, unsigned short, and unsigned long

Types `unsigned char`, `unsigned short`, and `unsigned long` are supported by the APW C Compiler and by many implementations of PCC, although they are not required by the basic C language definition. The VAX implementation of PCC and the APW C Compiler differ in the way they evaluate expressions involving these types. For example, the negation operator subtracts an `unsigned short` from 2^{16} under PCC and from 2^{32} under APW C.

Bit fields

APW C does not support signed bit fields. In the following example, implementations using unsigned bit fields will set `i` to 3:

```
struct {unsigned int field:2;}  x;
x.field  =  3;
i  =  x.field;
```

Evaluation order

APW C does not define the evaluation order of certain expressions. Expressions with side effects, such as function calls and the ++ and -- operators, may yield different results on different machines or with different compilers. Specifically, when a variable is modified as a side effect of an expression's evaluation and when the variable is also used at another point in the same expression, the value used may be either the value before modification or the value after modification.

Programs that rely on the order of evaluation in these situations are in error. The function call

```
f(i,i++)
```

is an example of an expression whose value is undefined.

String substitutions in #define statements

APW C, like MPW C, does not do string substitutions in `#define` macros , so

```
#define show(x) printf("x is %d\n", x)
i = 1; show(i);
```

will produce the output

```
x is 1
```

and not the output

```
i is 1
```

That is, the "`x is %d\n`" string is never modified by expansion of a `#define` macro.

Assignment operators

The *op*= form of assignment operators may not have a space or comment between the *op* and =, as shown in this example,

```
i +   /* APW C will choke on this. */   = 1;
```

Language anachronisms

Several constructs formerly considered part of the C language are now considered anachronisms. The compiler considers these constructs to be invalid. The anachronisms are described as follows.

Assignment operators

The =*op* form of assignment operators is not supported. Alternative interpretations are accepted without warning. In particular,

```
x =-  5;     is interpreted as     x =   (-5);
x =*  5;     is interpreted as     x =   (*5);
x =&  p;     is interpreted as     x =   (&p);
```

Initialization

The equal sign that introduces an initializer must be present. The anachronism

```
int i 1;
```

is considered an error.

Compiler limitations

On the Apple IIGS, the total size of all declared global scalar variables, static scalar variables, and scalar constants cannot exceed 64K because they are accessed using short addressing. Aggregate types (structures, arrays, and string constants) are stored in a separate large memory segment and accessed with long addressing. Their size is effectively limited only by available memory.

Automatic variables are limited by the available stack space, which can never exceed 32K.

Each code segment is limited to 64K.

Due to a limitation of ProDOS, only six levels of `#include` can be nested.

Performance tips

The following practices improved performance:

- Use `unsigned` types whenever possible. (Doing this improves performance markedly.)
- Declare `auto` aggregate variables after all `auto` scalars. (Doing this improves performance markedly.)
- Declare `auto` pointers before other `auto` variables.

The segment command

You can use the `segment` command to create load segments. The command

```
segment "segname"[, dynamic]
```

can only appear between functions: it assigns all objects that follow it, up to the next `segment` command or the end of file, to the load segment named "*segname*". (Note that the quotation marks are required.)

By default, this command creates a static load segment. The `dynamic` option creates a dynamic segment.

The `segment` command can be used to split up a code segment that would be larger than 64K.

The #append directive

The APW C preprocessor processes the usual directives, as well as one that is peculiar to APW C:

```
#append "filename"
```

When this directive is used, it must appear between functions: the variable *filename* is the name of the next file in the compilation sequence. This directive normally appears at the end of a file, as everything after it will be ignored. It should not appear in an include file.

START.ROOT, restartability, and StandAlone

`START.ROOT` is normally the first file linked into your application; that is, the first `LINK` command begins with

```
(LINK 2/START ...)
```

`START.ROOT` is responsible for initializing SANE and setting up `argc` and `argv` parameters to `main()`. `START.ROOT` then calls `main()`. When your program terminates—that is, when program control is returned from your `main()` procedure or when `exit()` is called—`START.ROOT` closes any files opened via the standard C library `open` call and then returns control either to ProDOS 16 or to APW.SYS16, depending on which one launched the program. Thus, you don't have to make a ProDOS 16 `Quit` call explicitly: `START.ROOT` does it for you.

When your program terminates, the integer variable `qFlag` determines whether or not the program will be recognized by ProDOS 16 as restartable. The variable `qPath` determines which program is to be launched next. See the Apple IIGS ProDOS 16 Reference about the `QUIT` call. The variables are declared as

```
extern int qFlag;
extern char *qPath;
```

The default setting of these variables is such that the program is not restartable (the *restart-from-memory* flag bit of `qFlag` is off), no new program is launched, and control will not return to the application (the *return* flag of `qFlag` is off).
APW always ignores this information. APW will only recognize your program as being restartable if your program appears in a line in the `SYSCMND` file, with an asterisk in the second column. (For more information, see the SYSCMND section of the *APW Reference*.) ProDOS will pay attention to this information.

The variable `extern int StandAlone` is declared by the Standard C Library. When an APW C program is started, this variable is set to zero (false) if the program is not a standalone program (that is, running under APW) or set to nonzero (true) if running as a standalone program (that is, an application).

Code-generation memory model

The memory model used by the code generation is a mixed model, intended to most effectively exploit the architecture of the 65816, which has addressing modes that deal with memory in a linear fashion, and others that treat memory as being divided into segments.

Essentially, long, or linear, addressing is used for all pointer values: pointers are 32-bit values, which contain 24-bit machine addresses. Global scalar variables, however, are referenced internally by using the more efficient 16-bit addressing modes. For these operations, the high byte of the 24-bit address is derived from the processor's data bank register, which is initialized by the `START.ROOT` module to point to the bank in which the load segment that contains the global data has been loaded. This feature is the reason that total global scalar storage is limited to 64K. On the other hand, global arrays and structures, are always addressed using long addressing, so it is possible to have more than 64K of array space. `Struct`'s and `union`'s are accessed using indexed addressing, so they are limited in size to 64K. Array references will use the faster 16-bit indexed addressing modes if the array is less than 64K in size.

Local variables (`auto`) are allocated on the 65816 machine stack. The machine stack pointer is a 16-bit register; the bank address of the stack is always bank 0. Thus, the maximum stack size is limited to a theoretical 64K: in practice, this size is considerably smaller due to competing use of bank-0 memory by the system and other potentially resident programs.

The start code initializes a default stack size of 4K, using this code from CLIB.

```
StackMin        START  Direct
        KIND    $12
        ds      $1000  * this is tne amount of stack
        END
```

You can override this default by modifying this code, assembling it, and linking it in explicitly, as shown in the sample application BONES.

Storage for local variables is created dynamically on the stack upon function entry. If less than 256 bytes are required for parameter storage, internal temporary variables, and local variables, then all local variables will be addressed via direct page addressing, and pointer dereferencing using local variables will generally use indirect long addressing. If more than 256 bytes are required, the compiler will have to use indexed addressing to access variables that extend beyond the first 256 bytes of stack storage allocated. The first declared variables are the first allocated, so declaring your frequently used local variables first will guarantee that the most efficient addressing modes will be used in referencing them.

All function calls are made via long subroutine calls.

If you are writing a Pascal-style C function to be called by the ROM (for example, a DefProc), and if you want to reference statically allocated scalar variables (which the C compiler puts in the load segment `~globals`), your function should begin with a call to `SaveDB`, which saves the value in the data bank register and changes the data bank register to point to `~globals`. Your function should end with a call to `RestoreDB`, which restores the data bank register to the value it held before `SaveDB` was called. References to statically allocated array variables (which the C compiler puts in the load segment `~arrays`) use 24-bit addressing and don't use the data bank register. If you are only using array variables and auto variables, you don't need to call `SaveDB` and `RestoreDB`. These two procedures are provided as part of the source code of the sample desk accessory.

An easy way to make a variable reside in `~arrays` instead of `~globals` is to declare it as a one-element array. For example, use

```
char c[1]; int n[1]; double x[1];
```

instead of

```
char c; int n; double x;
```

Chapter 5

The Standard C Library

About the Standard C Library

This chapter describes the Standard C Library provided with APW C. The Standard C Library is a collection of basic routines that let you read and write files, examine and manipulate strings, perform data conversion, acquire and release memory, and perform mathematical operations.

The chapter begins with an introduction to the error-number conventions used in the Standard C Library, followed by the library functions and macros arranged alphabetically under the name of the header file that contains them. Each header file contains a group of related functions or macros. For example, both the `fread` and `fwrite` macros are found under the `fread` header. All of the function names and other identifiers used in Standard C Library routines are listed in Appendix D, "Library Index." To find out where in this chapter a particular identifier is described, consult Appendix D.

❖ *Note:* Remember that identifiers in C are case-sensitive and should be spelled exactly as shown in the synopsis. Filenames (as in `#include` statements) are not case-sensitive. By convention, they are written in uppercase.

The library routines under each header are documented as follows:

- *Synopsis* shows the code you need to add to your program when using these library routines and the files you need to include at compile time.
- *Description* discusses the library routines and their input and output.
- *Diagnostics* describes error conditions.
- *Return value* describes the values returned by the routines.
- *Example* contains examples of commands.
- *Note* contains remarks.
- *Warning* gives cautions.
- *See also* provides the names of other library routines or sections in this chapter related to the ones described in the current section. It may also provide references to other Apple manuals, such as the *Apple Numerics Manual* or the *Apple IIGS Toolbox Reference*.

Not all of these will be found under each header.

❖ *Note:* Specific support for desk accessories has not been a consideration in the design of this library.

Important

Many of the functions in the Standard C Library use parameters of type `int`—this is necessary to achieve compatibility with other implementations of the Standard C Library. On the Apple IIGS, type `int` is 16 bits rather than 32, so any parameter of type `int` is limited to the range 0 to 65,535. A C program designed to use parameters of type `int` to pass values greater than 65,535 will generate no compiler error, but it will not work correctly under APW C.

Error numbers

Synopsis

```
#include <ERRNO.H>

extern int errno;
```

Description

Many of the Standard C Library functions have one or more possible error returns. An otherwise meaningless return value, usually –1, indicates an error condition: see the descriptions of individual functions for details. The external variable `errno` also provides an error number. The variable `errno` is only valid immediately after a call; it is not cleared on successful calls, so it should be tested only if the return value indicates an error.

The error name appears in brackets following the text in a library function description: for example,

The next attempt to write a nonzero number of bytes will signal an error.

[ENOSPC]

Not all possible error numbers are listed for each library function because many errors are possible for most of the calls. Some UNIX operating system error numbers do not apply to the Apple IIGS and are not documented in this manual. Some calls go to the Apple IIGS ROM, and as a side effect return a value in `_toolErr` in addition to the value in `errno`. Some calls, such as `printf` and `scanf`, may change these global variables even when they succeed.

Here is a list of the error numbers that can be returned in `errno` and their names as defined in the `ERRNO.H` file.

2 `ENOENT` *No such file or directory*
A file whose filename is specified does not exist, or one of the directories in a pathname does not exist.

5 `EIO` *I/O error*
A physical I/O error has occurred. In some cases, this error may be signaled on a call following the one to which it actually applies.

6 `ENXIO` *No such device or address*
An I/O operation on a particular file refers to a subdevice that does not exist, or the I/O operation is beyond the limits of the device. This error may also occur when, for example, no disk is present in a drive.

9 `EBADF` *Bad file number*
Either a file descriptor does not refer to an open file, or a read (or write) request has been made to a file that is open only for writing (or reading).

12 `ENOMEM` *Not enough space*
The system ran out of memory while the library call was executing.

13 `EACCES` *Permission denied*
An attempt was made to access a file in a way forbidden by the protection system.

14 `EFAULT` *Bad pathname*
A supplied pathname has incorrect syntax.

16 `EBUSY` *Device or resource busy*
Two or more online volumes have identical volume names.

17 EEXIST *File exists*
An existing file was mentioned in an inappropriate context: for example, `open(file, O_CREAT|O_EXCL)`.

19 ENODEV *No such device*
An attempt was made to apply an inappropriate system call to a device: for example, an attempt was made to read from a write-only device.

20 ENOTDIR *Not a directory*
An object that is not a directory was specified where a directory is required: for example, in a pathname prefix.

22 EINVAL *Invalid parameter*
An invalid parameter was provided to a library function.

23 ENFILE *File table full*
The system's table of open files is full, so temporarily a call to open cannot be accepted.

24 EMFILE *Too many open files*
The system cannot allocate memory to record another open file.

26 ETXTBSY *Text file busy*
An attempt has been made to perform a disallowed operation on an open file.

28 ENOSPC *No space left on device*
A write operation to an ordinary file cannot be performed because the device has no free space left.

30 EROFS *Read-only file system*
An attempt to modify a file or directory was made for a device mounted for read-only access.

Note

Calls that interface with the Apple IIGS I/O system (such as `open`, `close`, `read`, `write`, and `ioctl`) can set the external variable `_toolErr` as well as `errno` on errors. This is a side effect: it is not safe to assume any relationship between the error number returned in `errno` and the number that may be returned in `_toolErr`. To detect errors in Standard C Library calls, use `errno`; to detect errors in Toolbox calls use `_toolErr`.

This section documents the values returned in `errno`. The Toolbox errors returned in `_toolErr` are documented in the chapter "The System Error Handler" of the *Apple IIGS Toolbox Reference*.

abs—return integer absolute value

Synopsis

```
int abs(i)
  int i;
```

Description

Function abs returns the absolute value of `i`.

Note

The absolute value of the negative integer with the largest magnitude is undefined.

See also

`floor`

atof—convert ASCII string to floating-point number

Synopsis

```
#include <MATH.H>

extended atof(str)
  char *str;
```

Description

Function `atof` converts a character string pointed to by `str` to an extended-precision floating-point number. The first unrecognized character ends the conversion. Function `atof` recognizes an optional string of white-space characters (spaces or tabs), then an optional sign, then a string of digits optionally containing a decimal point, and then an optional *e* or *E* followed by an optionally signed integer. If the string begins with an unrecognized character, `atof` returns a NaN.

Function `atof` recognizes `"INF"` as infinity and `"NAN"` (optionally followed by parentheses that may contain a string of digits) as a NaN, with NaN code given by the string of digits. Case is ignored in the infinity and NaN strings.

Diagnostics

Function `atof` honors the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

`scanf`

"Conversions Between Decimal Formats" in Chapter I-4 of the *Apple Numerics Manual*

atoi—convert string to integer

Synopsis

```
#include <STDLIB.H>

int atoi(str)
  char *str;
long atol(str)
  char *str;
```

Description

The character string `str` is scanned up to the first nondigit character other than an optional leading minus sign (–). Leading white-space characters (spaces and tabs) are ignored.

A plus sign (+) is considered a nondigit character.

Return value

Function `atoi` returns as an integer the decimal value represented by `str`.
Function `atol` returns as a long integer the decimal value represented by `str`.

Note

Overflow conditions are ignored.

See also

`atof`, `scanf`, `strtol`

close—close a file descriptor

Synopsis

```
int close(fildes)
  int fildes;
```

Description

Parameter `fildes` is a file descriptor obtained from an `open`, `creat`, `dup`, or `fcntl` call. Function `close` closes the file descriptor indicated by `fildes`.

Function `close` fails if `fildes` is not a valid open file descriptor. [`EBADF`]

Diagnostics

Upon successful completion, this function returns a value of 0. Otherwise, it returns a value of –1 and sets `errno` to indicate the error.

See also

`creat`, `dup`, `fcntl`, `open`

conv—translate characters

Synopsis

```
#include <CTYPE.H>

int toupper(c)
  int c;
int tolower(c)
  int c;
int _toupper(c)
  int c;
int _tolower(c)
  int c;
int toascii(c)
  int c;
```

Description

Functions `toupper` and `tolower` have as their domain the set of ASCII characters (0 through 127) and the constant `EOF` (–1). If parameter `c` to `toupper` represents a lowercase letter, the result is the corresponding uppercase letter. If parameter `c` to `tolower` represents an uppercase letter, the result is the corresponding lowercase letter. All other parameters in the domain are returned unchanged.

Macros `_toupper` and `_tolower` produce the same results as functions `toupper` and `tolower`, but have restricted domains and are faster. Macro `_toupper` requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. Macro `_tolower` requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Parameters outside the domain cause undefined results.

Macro `toascii` converts `c` by clearing all bits that are not part of a standard ASCII character. It is used to achieve compatibility with other systems.

Note

These routines do not support the Apple IIGS extended character set (with values greater than 0x7F). For values outside the stated domain, the result is undefined.

See also

`ctype`, `getc`

creat—create a new file or rewrite an existing file

Synopsis

```
int creat(filename)
  char *filename;
```

Description

Function `creat` creates a new file or prepares to rewrite an existing file, `filename`. If the file exists, its length is set to 0.

Function `creat(filename)` is equivalent to

```
open(filename,O_WRONLY|O_TRUNC|O_CREAT)
```

Upon successful completion, a nonnegative integer (the file descriptor) is returned and the file is open for writing. The file pointer is set to the beginning of the file. A maximum of about 30 files may be open at a given time; the actual maximum depends upon the current system environment.

Return value

Upon successful completion, this function returns a nonnegative integer (the file descriptor). Otherwise, it returns a value of –1 and sets `errno` to indicate the error.

Note

Other implementations of `creat` specify a second parameter, `mode`. This version ignores any second parameter.

See also

`close`, `open`

ctype—classify characters

Synopsis

```
#include <CTYPE.H>

int isalpha(c)
  int c;
int isupper(c)
  int c;
int islower(c)
  int c;
int isdigit(c)
  int c;
int isxdigit(c)
  int c;
int isalnum(c)
  int c;
int isspace(c)
  int c;
int ispunct(c)
  int c;
int isprint(c)
  int c;
int isgraph(c)
  int c;
int iscntrl(c)
  int c;
int isascii(c)
  int c;
```

Description

These macros classify character-coded integer values by table lookup, returning nonzero for `true` and zero for `false`. Macro `isascii` is defined for all integer values; the other macros are defined only where `isascii` is true and for the single non-ASCII value EOF (–1).

Macro	Returns true if
`isascii`	c is an ASCII character code less than octal 0200.
`isalpha`	c is a letter [A–Z] or [a–z].
`isupper`	c is an uppercase letter [A–Z].
`islower`	c is a lowercase letter [a–z]
`isdigit`	c is a digit [0–9].
`isxdigit`	c is a hexadecimal digit [0–9], [A–F], or [a–f].
`isalnum`	c is alphanumeric (letter or digit).
`isspace`	c is a space, tab, return, newline, vertical tab, or form-feed character.
`ispunct`	c is a punctuation character (neither control nor alphanumeric).
`isprint`	c is a printing character in the range space (octal 040) through tilde (octal 0176).
`isgraph`	c is a printing character, similar to `isprint` except that it is false for space.
`iscntrl`	c is a delete character (octal 0177) or an ordinary control character (less than octal 040).

Warning If c is not in the domain of the function, the result is undefined.

Note These macros do not support the Apple IIGS extended character set. For values outside the domain, the result is undefined.

dup—duplicate an open file descriptor

Synopsis

```
int dup(fildes)
  int fildes;
```

Description

Function dup returns a new file descriptor with these features:

- It refers to the same open file as the original descriptor.
- It shares the original descriptor's file pointer.
- It has the same access mode (that is, read, write, or read/write) as the original descriptor.

Parameter `fildes` is a file descriptor obtained from an `open`, `creat`, `dup`, or `fcntl` call. The new file descriptor returned by dup is the lowest one available.

The function call `dup(fildes)` is equivalent to

```
fcntl(fildes, F_DUPFD, 0)
```

Function dup fails if parameter `fildes` is not a valid open file descriptor. [EBADF]

Return value

Upon successful completion, this function returns a nonnegative integer (the file descriptor) is returned. Otherwise, it returns a value of –1 and sets `errno` to indicate the error.

See also

`close`, `fcntl`, `open`

ecvt—convert a floating-point number to a string

Synopsis

```
#include <MATH.H>

char *ecvt(value, ndigit, decpt, sign)
  extended value;
  int ndigit, *decpt, *sign;
char *fcvt(value, ndigit, decpt, sign)
  extended value;
  int ndigit, *decpt, *sign;
```

Description

Function `ecvt` converts `value` to a null-terminated string of `ndigit` digits and returns a pointer to this string as the function result. The low-order digit is rounded.

The decimal point is not included in the returned string. The position of the decimal point is indicated by `decpt`, which indirectly stores the position of the decimal point relative to the returned string. If the `int` pointed to by `decpt` is negative, the decimal point lies to the left of the returned string. For example, if the string is "`12345`" and `decpt` points to an `int` of 3, the value of the string is 123.45; if `decpt` points to –3, the value of the string is .00012345.

If the sign of the converted value is negative, the `int` pointed to by `sign` is nonzero; if the sign is positive, it is zero.

Function `fcvt` provides fixed-point output in the style of Fortran F-format output. Function `fcvt` differs from `ecvt` in its interpretation of `ndigit`:

- In `fcvt`, `ndigit` specifies the number of digits to the right of the decimal point.
- In `ecvt`, `ndigit` specifies the number of digits in the string.

Note

The string pointed to by the function result is static data whose contents are overwritten by each call. To preserve the value, copy it before calling the function again.

See also

`printf`

"Conversions Between Decimal Formats" in Chapter I-4 of the *Apple Numerics Manual*

exit—terminate the current application

Synopsis

```
#include <STDLIB.H>

void exit(status)
  int status;
void _exit(status)
  int status;
```

Description

Functions `exit` and `_exit` close open file descriptors and terminate the application or tool. Here is the order in which `exit` performs its duties:

1. It executes all exit procedures, including the exit procedures for the Standard I/O Package if routines from that package were used, in reverse order of their installation by `onexit`. All buffered files are flushed and closed.
2. It closes all open files that were opened with `open` or `fopen`.
3. If the program is a tool running under the APW Shell, the `exit` function returns status and control information to the APW Shell by placing a return value in the APW variable `status` and terminating the application.

Function `_exit` circumvents the exit procedures described in step 1 just given. Use `_exit` instead of `exit` to abort your program when you are uncertain about the integrity of the data space.

Return value

The `main` program is a function that returns an integer. The return value of `main` is interpreted by the APW Shell as the program status. When you call `exit` or `_exit`, the `status` parameter is returned to the APW Shell as the return value for the application's main function. This value is 0 for normal execution or a nonzero value for errors (typically 1..3). A `main` program that returns to the shell without setting `status` to an integer value returns 0.

There is no return from `exit` or `_exit`.

Note

Functions `exit` and `_exit` do not close files you opened with calls to the I/O routines documented in the *Apple IIGS Toolbox Reference*.

See also

`onexit`, `stdio`

exp—exponential, logarithm, power, square-root functions

Synopsis

```
#include <MATH.H>

extended exp(x)
  extended x;
extended log(x)
  extended x;
extended log10(x)
  extended x;
extended pow(x, y)
  extended x, y;
extended sqrt(x)
  extended x;
```

Description

Function `exp(x)` returns e^x, where e is the natural logarithm base.

Function `log(x)` returns the natural logarithm of x, $\log_e x$.

Macro `log10(x)` returns the base-10 logarithm of x, $\log_{10} x$.

Macro `pow(x, y)` returns x^y.

Function `sqrt(x)` returns the square root of x.

For special cases, these functions return a NaN or signed infinity as appropriate.

Diagnostics

These functions honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

`hypot`, `sinh`

"Exception Flags and Halts" in Chapter I-8, and "Logarithm Functions" and "Exponential Functions" in Chapter I-10 of the *Apple Numerics Manual.*

faccess—named-file access and control

Synopsis

```
#include <FCNTL.H>

int faccess(filename, cmd, arg)
  char *filename;
  unsigned int cmd;
  char *arg;
```

Description

Function `faccess` provides access to control and status information for named files. (Compare with function `ioctl`, which provides different control and status information for open files.)

Parameter `cmd` must be set to one of the constants in the following list to indicate what operation is to be performed on the file. As noted in the list, some calls to `faccess` also require the `arg` parameter, which is usually as a pointer to a `char`.

The following commands are available to all programs.

Value of cmd	Description
`F_DELETE`	Deletes the named file, or returns an error if the file is open. The parameter `arg` is ignored.
`F_RENAME`	Renames the named file. The parameter `arg` is a pointer to a string containing the new name.
`F_TYPE`	Sets the type of the file to the value of the parameter `arg`.
`F_AUX`	Sets the auxiliary type of the file to the value of the parameter `arg`.

For example, `faccess(thing, F_TYPE, 0x04)` sets the type of file `thing` to $04, for ASCII text file. (A list of file types is in the *ProDOS 16 Technical Reference*.)

Return value

Upon successful completion, `faccess` returns a nonnegative value, which is usually 0. If the device for the named file cannot perform the requested command, `faccess` returns –1 and sets `errno` to indicate the error.

Note

The `cmd` value `F_OPEN` is reserved for operating-system use.

See also

`ioctl`, `unlink`

fclose—close or flush a stream

Synopsis

```
#include <STDIO.H>

int fclose(stream)
  FILE *stream;
int fflush (stream)
  FILE *stream;
```

Description

Function `fclose` closes a file that was opened by `fopen`, `freopen`, or `fdopen`. Function `fclose` causes any buffered data for `stream` to be written, and the buffer (if one was allocated by the system) to be released; `fclose` then calls `close` to close the file descriptor associated with `stream`. The value of the parameter `stream` cannot be used unless it is reassigned with `fopen`, `fdopen`, or `freopen`.

Function `fclose` fails if the file descriptor associated with `stream` is already closed. [`ENOENT`]

Function `fclose` is performed automatically for all open `FILE` streams when `exit` is called.

Function `fflush` causes any buffered data for `stream` to be written; `stream` remains open.

Return value

These functions return either 0 if the operation succeeds or `EOF` (-1) if an error is detected (such as trying to write to a file that has not been opened for writing).

See also

`close`, `exit`, `fopen`, `setbuf`

fcntl—file control

Synopsis

```
#include <FCNTL.H>

int fcntl(fildes, cmd, arg)
  int fildes;
  unsigned int cmd;
  int arg;
```

Description

Function `fcntl` is used for duplicating file descriptors. A file remains open until all of its file descriptors are closed.

Parameter `fildes` is an open file descriptor obtained from an `open`, `creat`, `dup`, or `fcntl` call. Parameter `cmd` takes the value `F_DUPFD`, which tells `fcntl` to return the lowest numbered available file descriptor greater than or equal to `arg`. Normally, `arg` is greater than or equal to 3, to avoid obtaining the standard file descriptors 0, 1, and 2. Function `fcntl` returns a new file descriptor that points to the same open file as `fildes`. The new file descriptor has the same access mode (read, write, or read/write) and file pointer as `fildes`. Any I/O operation changes the file pointer for all file descriptors that share it.

Function `fcntl` fails if one or more of the following are true:

- Parameter `fildes` is not a valid open file descriptor. [`EBADF`]
- Parameter `arg` is negative or greater than the highest allowable file descriptor. [`EINVAL`]

Return value

Upon successful completion, this function returns a new file descriptor. Otherwise, it returns a value of –1 and sets `errno` to indicate the error.

Note

The `F_GETFD`, `F_SETFD`, `F_GETFL`, and `F_SETFL` commands of `fcntl` are not supported on the Apple IIGS.

See also

`close`, `dup`, `open`

ferror—stream status inquiries

Synopsis

```
#include <STDIO.H>

int feof(stream)
  FILE *stream;
int ferror(stream)
  FILE *stream;
void clearerr(stream)
  FILE *stream;
int fileno(stream)
  FILE *stream;
```

Description

Macro `feof` returns a nonzero number when an end-of-file condition has previously been detected reading the named input stream; otherwise, it returns zero.

Macro `ferror` returns a nonzero number when an I/O error has previously occurred reading from or writing to the named stream; otherwise, it returns zero.

Macro `clearerr` resets the error and end-of-file indicators to zero on the named stream.

Macro `fileno` returns the integer file descriptor associated with the named stream. See `open`.

See also

`open`, `fopen`

floor—floor, ceiling, mod, absolute value functions

Synopsis

```
#include <MATH.H>

extended floor(x)
  extended x;
extended ceil(x)
  extended x;
extended fmod(x, y)
  extended x, y;
extended fabs(x)
  extended x;
```

Description

Function `floor(x)` returns the largest integer (as an extended-precision number) not greater than `x`.

Function `ceil(x)` returns the smallest integer not less than `x`.

Whenever possible, `fmod(x, y)` returns the number f with the same sign as `x`, such that $x = iy + f$ for some integer i, and $|f| < |y|$. If `y` is 0, `fmod` returns a NaN.

Function `fabs(x)` returns $|x|$, the absolute value of `x`.

See also

`abs`

"Round to Integer Value" in Chapter I-3 and "Rounding Direction" in Chapter I-8 of the *Apple Numerics Manual*

fopen—open a buffered file stream

Synopsis

```
#include <STDIO.H>

FILE *fopen(filename, type)
  char *filename, *type;
FILE *freopen(filename, type, stream)
  char *filename, *type;
  FILE *stream;
FILE *fdopen(fildes, type)
  int fildes;
  char *type;
```

Description

Function `fopen` opens the file named by `filename` and associates a stream with it. Function `fopen` returns a pointer to the `FILE` structure associated with the stream.

Parameter `filename` points to a character string that contains the name of the file to be opened. The `filename` parameter cannot be the pseudo-filename `.printer`, a device name (such as `.D2`), or the double period (`..`).

Parameter `type` points to a character string consisting of one of the string values in the first column in the following table. The remaining columns explain how `type` is used. (For more information, see `open`.)

Value	Open mode used	Description
`r`	`O_RDONLY`	Open for reading only.
`w`	`O_WRONLY|O_CREAT|O_TRUNC`	Truncate or create for writing.
`a`	`O_WRONLY|O_CREAT|O_APPEND`	Append: open for writing at end of file, or create for writing.
`r+`	`O_RDWR`	Open for update (reading and writing).
`w+`	`O_RDWR|O_CREAT|O_TRUNC`	Truncate or create for updating.
`a+`	`O_RDWR|O_CREAT|O_APPEND`	Append: open or create for updating at end of file.

When a file is written to a device, normally certain characters are translated to match the needs of the device or the expectations of ProDOS for a normal text file (such as translating `\n` to `CR` rather than `LF`). The following values, with `b` added to the string, suppress such translations:

Value	Open mode used	Description
`rb`	`O_RDONLY|O_BINARY`	Open for reading only.
`wb`	`O_WRONLY|O_CREAT|O_TRUNC|O_BINARY`	Truncate or create for writing.
`ab`	`O_WRONLY|O_CREAT|O_APPEND|O_BINARY`	Append: open for writing at end of file, or create for writing.
`rb+`	`O_RDWR|O_BINARY`	Open for update (reading and writing).
`wb+`	`O_RDWR|O_CREAT|O_TRUNC|O_BINARY`	Truncate or create for updating.
`ab+`	`O_RDWR|O_CREAT|O_APPEND|O_BINARY`	Append: open or create for updating at end of file.

❖ *Note:* The `b` and the `+` can be reversed.

Function freopen substitutes the file named by filename for the open stream. The original stream is closed, regardless of whether the open operation ultimately succeeds. Function freopen returns a pointer to the FILE structure associated with stream. Function freopen is typically used to attach the previously opened streams associated with stdin, stdout, and stderr to other files. The filename parameter cannot be the pseudo-filename .printer, a device name (such as .D2), or the double period (..).

Function fdopen associates a stream with a file descriptor by formatting a file structure from the file descriptor. Thus, fdopen can be used to access the file descriptors returned by open, creat, dup, and fcntl. (These calls return file descriptors, and not pointers to a FILE structure.) The type of the stream must agree with the mode of the open file.

When a file is opened for updating, both input and output operations may be performed on the resulting stream. However, an output operation may not be directly followed by an input operation without an intervening fseek or rewind, and an input operation may not be directly followed by an output operation without an intervening fseek or rewind, or without an input operation that encounters an end-of-file condition.

When a file is opened for appending (that is, when type is a or a+), it is impossible to overwrite information already in the file. The function fseek may be used to reposition the file pointer to any position in the file, but when output is written to the file the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output.

Return values If they succeed, the functions fopen, freopen, and fdopen return a valid file pointer. If they fail, they return NULL.

The maximum number of open FILE streams is NFILE (defined in STDIO.H, currently 20). The maximum number of open disk files may be less than NFILE, as determined by the current release of ProDOS. (ProDOS 16, Version 1.0, permits 8 open disk files; later releases may increase this number.)

Note The parameter type must have one of the values in the first column in the table; do not use values intended for open, such as O_RDONLY.

See also open, fclose, fseek

fread—binary input/output

Synopsis

```
#include <STDIO.H>

int fread(ptr, size, nitems, stream)
  char *ptr;
  int size, nitems;
  FILE *stream;
int fwrite(ptr, size, nitems, stream)
  char *ptr;
  int size, nitems;
  FILE *stream;
```

Description

Function `fread` copies `nitems` items of data from the named input stream into an array beginning at `ptr`. An item of data is a sequence of `size` bytes (not necessarily terminated by a null byte). Function `fread` stops appending bytes if an end-of-file or error condition is encountered while reading `stream` or if `nitems` items have been read. Function `fread` leaves the file pointer in `stream` pointing to the byte following the last byte read.

Function `fwrite` writes up to `nitems` items of data to the named output stream from the array pointed to by `ptr`. An item is a sequence of `size` bytes. Function `fwrite` stops writing when it has written `nitems` items of data or if it encounters an error condition on `stream`. Function `fwrite` does not change the contents of the array pointed to by `ptr`.

The parameter `size` is typically

```
sizeof(*ptr)
```

where `sizeof` specifies the length of an item pointed to by `ptr`. If `ptr` points to a data type other than `char`, it should be cast into a pointer to `char`.

Return values

The functions `fread` and `fwrite` return the number of items read or written. If `nitems` is 0 or negative, no characters are read or written, and both `fread` and `fwrite` return 0.

See also

`fopen`, `getc`, `gets`, `printf`, `putc`, `puts`, `read`, `scanf`, `stdio`, `write`

frexp—manipulate parts of floating-point numbers

Synopsis

```
#include <MATH.H>

extended frexp(value, eptr)
  extended value;
  int *eptr;
extended ldexp(value, exp)
  extended value;
  int exp;
extended modf(value, iptr)
  extended value, *iptr;
```

Description

Every nonzero number can be written uniquely as $x \cdot 2^n$, where the mantissa (fraction) x is in the range $0.5 \leq |x| < 1.0$ and the exponent n is an integer. Function `frexp` returns the mantissa of an `extended` value and stores the exponent indirectly in the location pointed to by `eptr`. Note that the mantissa here differs from the significant described in the *Apple Numerics Manual,* whose normal values are in the range $1.0 \leq |x| < 2.0$.

Function `ldexp` returns the quantity `value` * 2^{exp}.

Function `modf` returns the signed fractional part of `value` and stores the integral part indirectly in the location pointed to by `iptr`.

Diagnostics

Function `ldexp` honors the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

"Binary Scale and Log Functions" in Chapter I-9 of the *Apple Numerics Manual*

fseek—reposition a file pointer in a stream

Synopsis

```
#include <STDIO.H>

int fseek(stream, offset, whence)
  FILE *stream;
  long offset;
  int whence;
void rewind(stream)
  FILE *stream;
long ftell(stream)
  FILE *stream;
```

Description

Function `fseek` sets the position of the next input or output operation on the stream. The new position is `offset` bytes from the beginning, the current position, or the end of file when the value of `whence` is 0, 1, or 2, respectively. If `whence` is 1 or 2, `offset` may be negative.

The call

```
rewind(stream)
```

is equivalent to

```
fseek(stream, 0L, 0)
```

except that no value is returned.

Functions `fseek` and `rewind` undo any effects of `ungetc` if the new location is not within the same buffer.

After `fseek` or `rewind`, the next operation on a file opened for updating may be either input or output.

Function `ftell` returns the offset of the current byte relative to the beginning of the file associated with the named stream.

Diagnostics

Function `fseek` returns a nonzero number for improper seek operations; otherwise it returns zero. An example of an improper seek operation is an `fseek` before the beginning of a file.

See also

`lseek`, `fopen`, `ungetc`

getc—get a character or a word from a stream

Synopsis

```
#include <STDIO.H>

int getc(stream)
  FILE *stream;
int getchar()
int fgetc(stream)
  FILE *stream;
int getw(stream)
  FILE *stream;
```

Description

Macro `getc` returns the next character from the named input stream. It also moves the file pointer, if defined, ahead one character in `stream`. Macro `getc` cannot be used if a function is necessary; for example, you cannot have a function pointer point to it. Macro `getc` returns the integer `EOF` whenever an end-of-file or error condition occurs.

Macro `getchar` returns the next character from the standard input stream, `stdin`.

Function `fgetc` produces the same result as macro `getc`; function `fgetc` runs more slowly than macro `getc` but takes less space per invocation. Also, you can have a pointer to `fgetc` but not to `getc`.

Function `getw` returns the next `int` (that is, 2 bytes) from the named input stream so that the order of bytes in the stream corresponds to the order of bytes in memory. Function `getw` returns the constant `EOF` upon encountering an end-of-file or error condition. Because `EOF` is a valid integer value, `feof` and `ferror` should be used to check the success of `getw`. Function `getw` increments the associated file pointer, if defined, to point to the next `int`. Function `getw` assumes no special alignment in the file.

Return values

These calls either return data from the stream or return the integer constant `EOF` (–1) when an end-of-file or error condition occurs.

Note

Because it is implemented as a macro, `getc` treats a stream parameter with side effects incorrectly. In particular,

```
getc(*f++)
```

doesn't work as you would expect. Instead use

```
fgetc(*f++)
```

See also

`ferror`, `fopen`, `fread`, `gets`, `scanf`, `stdio`

getenv—access exported APW Shell variables

Synopsis

```
#include <STDLIB.H>

char *getenv(varname)
  char *varname;
```

Description

The **environment** is the set of exported variables provided by the APW Shell. Function `getenv` provides access to variables in this set. (See "Variables" in Chapter 4 of the *Apple IIGS Programmer's Workshop Reference* for the list of standard exported shell variables.)

Function `getenv` searches the environment for a shell variable with the name specified by `varname`, and returns a pointer to the character string containing its value. The null pointer is returned if the shell variable is not defined or has not been exported. The shell-variable name search is case-insensitive.

Return value

Upon successful completion, this function returns a pointer to the value of `varname`. If the shell variable is not defined or not exported, the function returns a pointer to a null string.

For standalone applications, which do not run under the APW Shell, `getenv` always returns the null pointer.

Warning

Function `getenv` returns a pointer to the place in memory where a copy of the APW Shell variable resides. Do not modify the value of a shell variable in such a way as to increase its length.

gets—get a string from a stream

Synopsis

```
#include <STDIO.H>

char *gets(str)
  char *str;
char *fgets(str, maxlen, stream)
  char *str;
  int maxlen;
  FILE *stream;
```

Description

Function `gets` reads characters from the standard input stream `stdin` into the array pointed to by `str` until a newline character is read or until the end of file is reached. The newline character is discarded, and the string is terminated with a null (`\0`) character.

Function `fgets` reads characters from `stream` into the array pointed to by `str` until `maxlen-1` characters are read, a newline character is read and transferred to `str`, or the end of file is reached. The string is then terminated with a null character.

Return values

If the end of file is reached and no characters have been read, no characters are transferred to `str`, and `NULL` is returned. If a read error occurs, `NULL` is returned. If not, `str` is returned. (A read error will occur, for example, if you attempt to use these functions on a file that has not been opened for reading.)

Note

The array pointed to by `str` is assumed to be large enough; overflow is not checked.

The function `gets` omits the newline character in the string; `fgets` leaves it in.

See also

`ferror`, `fopen`, `fread`, `getc`, `scanf`, `stdio`

hypot—Euclidean distance function

Synopsis

```
#include <MATH.H>

extended hypot(x, y)
  extended x, y;
```

Description

Function `hypot` returns

```
sqrt (x * x + y * y)
```

taking precautions against unwarranted overflows.

Diagnostics

Function `hypot` honors the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

`exp`

"Exception Flags and Halts" in Chapter I-8 of the *Apple Numerics Manual*

ioctl—control a device

Synopsis

```
#include <IOCTL.H>

int ioctl(fildes, cmd, arg)
  int fildes;
  unsigned int cmd;
  long *arg;
```

Description

Function `ioctl` communicates with a file's device driver by sending control information, requesting status information, or both. Parameter `cmd` indicates which device-specific operations `ioctl` must perform. Here are the control values:

Value of cmd	Description
`FIOINTERACTIVE`	Function `ioctl` returns 0 if the device is interactive; it returns –1 and sets `errno` to `EINVAL`, if not parameter `arg` is ignored.
`FIOBUFSIZE`	Function `ioctl` returns the optimal buffer size for this device, in bytes; the buffer size is returned in a `long` variable pointed to by `arg`. If the device has no default buffer size, `ioctl` returns –1 and sets `errno` to `EINVAL`.
`FIOREFNUM`	Function `ioctl` returns the ProDOS file reference number associated with `fildes`; the reference number is returned in the `short` pointed to by `arg`. If `fildes` is not open on a ProDOS device (such as the console device), `ioctl` returns –1.
`FIOGETEOF`	Function `ioctl` stores the logical end of file in the `long` variable pointed to by `arg`. The value of `arg` is the size of the file, in bytes.
`FIOSETEOF`	Function `ioctl` sets the logical end of file specified in the `long` variable pointed to by `arg`. The value of `arg` is the new size of the file, in bytes. This command can be used to reduce or increase the size of the open file. The current file pointer is not affected unless the file size is set to a number less than the file pointer value.
`FIOGETMARK`	Function `ioctl` stores the logical file position specified in the `long` variable pointed to by `arg`. The value of `arg` is the distance, in bytes, from the start of the file to the current position.
`FIOSETMARK`	Function `ioctl` sets the logical file position specified in the `long` variable pointed to by `arg`. The value of `arg` is the distance, in bytes, from the start of the file to the current position.

Function `ioctl` fails if one or both of the following conditions exist:

- File descriptor `fildes` is not valid or is not open. [`EBADF`]
- Parameters `cmd` or `arg` are not valid for the device handler associated with `fildes`. [`EINVAL`]

Diagnostics If an error has occurred, a value of –1 is returned and `errno` is set to indicate the error.

Note For `cmd` values `FIOINTERACTIVE` and `FIOBUFSIZE`, a function return of –1 is a meaningful response, and not an error. For `FIOINTERACTIVE`, `errno` is set to `EINVAL` for devices that are not interactive. For `FIOBUFSIZE`, `errno` is set to `EINVAL` for devices that have no default buffering.

The `cmd` values `FIOLSEEK` and `FIODUPFD` are reserved for operating-system use.

Warning `FIOREFNUM` lets you perform ProDOS I/O operations (such as `SET_MARK`) that are not available through `ioctl`. Do not close or modify the file pointer using the reference number.

See also `fcntl`

lseek—move read/write file pointer

Synopsis

```
#include <FCNTL.H>

long lseek(fildes, offset, whence)
  int fildes;
  long offset;
  int whence;
```

Description

A file descriptor, `fildes`, is returned from a call to `creat`, `dup`, `fcntl`, or `open`. Function `lseek` sets the file pointer associated with `fildes` as follows:

- If `whence` is 0, the pointer is set to `offset` bytes. (The value of `offset` may be zero or positive.)
- If `whence` is 1, the pointer is set to its current location plus `offset`. (The value of `offset` may be negative, zero, or positive.)
- If `whence` is 2, the pointer is set to the size of the file plus `offset`. (The value of `offset` may be negative, zero, or positive.)

Upon successful completion, this function returns the file pointer value, measured in bytes from the beginning of the file.

The file pointer remains unchanged and `lseek` fails if one or more of the following is true:

- File descriptor `fildes` is not open. [`EBADF`]
- Parameter `whence` is not 0, 1, or 2. [`EINVAL`]
- The resulting file pointer would point before the beginning of the file. [`EINVAL`]

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

Return value

Upon successful completion, this function returns a nonnegative long integer indicating the file pointer value. Otherwise, it returns a value of –1 and sets `errno` to indicate the error.

Note

In previous versions of the Standard C Library, `tell(fildes)` was a function that returned the current file position. It is equivalent to the call

```
lseek(fildes, 0L, 1)
```

Warning

Function `lseek` has no effect on a file opened with the `O_APPEND` flag because the next write operation to the file always repositions the file pointer to the end of file before writing begins.

See also

`fseek`, `open`

malloc—memory allocator

Synopsis

```
#include <MALLOC.H>

char *malloc(size)
  unsigned int size;
char *lmalloc(size)
  unsigned long size;
void free(ptr)
  char *ptr;
char *realloc(ptr, size)
  char *ptr;
  unsigned int size;
char *calloc(nelem, elsize)
  unsigned int nelem, elsize;
void cfree(ptr, nelem, elsize)
  char *ptr;
  unsigned int nelem, elsize;
```

Description

Functions `malloc` and `free` provide a simple general-purpose memory-allocation package. The storage area expands as necessary when `malloc` is called.

Function `malloc` allocates the first sufficiently large contiguous free space it finds, and returns a pointer to a block of at least `size` bytes suitably aligned for any use. It calls `NewHandle` (see the *Apple IIGS Toolbox Reference*) to get more memory from the system when there is no suitable space already free. Since `malloc` uses a `size` parameter of type `unsigned int`, it can allocate blocks no larger than 64K bytes. If `size` is 32K or larger, `lmalloc` is called.

Function `lmalloc` allocates the first sufficiently large contiguous free space it finds, and returns a pointer to a block of at least `size` bytes suitably aligned for any use. It calls `NewHandle` (see the *Apple IIGS Toolbox Reference*) to get more memory from the system when there is no suitable space already free. Since `lmalloc` uses a `size` parameter of type `long`, it can allocate blocks larger than 64K.

Function `free` takes a parameter that is a pointer to a block previously allocated by `malloc` or `lmalloc`. If its size is greater than 2K, it is returned to the system using `DisposeHandle`. Blocks smaller than that are cached by `malloc` for further allocation by `malloc` only. Undefined results occur if the space assigned by `malloc` is overrun, or if a random value is passed to `free`.

Function `realloc` changes the size of the block pointed to by `ptr` to `size` bytes, and returns a pointer to the (possibly moved) block. The contents are unchanged up to the lesser of the new and old sizes. If no free block of `size` bytes is available in the storage area, `realloc` asks `malloc` to enlarge the storage area by `size` bytes and then moves the data to the new space. If `ptr` is `NULL`, `realloc` is equivalent to `malloc`.

Function `calloc` allocates space for an array of `nelem` elements of size `elsize`. The resulting space allocated is filled with zeros.

Function `cfree`, like `free`, frees memory allocated by `calloc`; `cfree` is included for compatibility with other systems. Parameters `nelems` and `elsize` are ignored.

Diagnostics

Functions `malloc`, `lmalloc`, `realloc`, and `calloc` return `NULL` if there is no available memory, or if the storage area has been detectably corrupted by a program's storing data outside the bounds of a block. When this happens, the block pointed to by `ptr` may have been destroyed.

memory—memory operations

Synopsis

```
#include <MEMORY.H>

char *memccpy(dest, source, c, n)
  char *dest, *source;
  int c, n;
char *memchr(source, c, n)
  char *source;
  int c, n;
int memcmp(a, b, n)
  char *a, *b;
  int n;
char *memcpy(dest, source, n)
  char *dest, *source;
  int n;
char *memset(dest, c, n)
  char *dest;
  char c;
  int n;
```

Description

These functions operate efficiently on memory areas (arrays of characters bounded by a count, rather than terminated by a null character). They do not check for the overflow of any receiving memory area.

Function `memccpy` copies characters from memory area `source` into `dest`, stopping after the first occurrence of character `c` has been copied or after `n` characters have been copied, whichever comes first. The function returns either a pointer to the character after the copy of `c` in `dest`, or `NULL` if `c` was not found in the first `n` characters of `source`.

Function `memchr` returns either a pointer to the first occurrence of character `c` in the first `n` characters of memory area `source`, or `NULL` if `c` does not occur.

Function `memcmp` compares its parameters, `a` and `b`, looking at the first `n` characters only. It returns an integer less than, equal to, or greater than 0, depending on whether `a` is less than, equal to, or greater than `b`, respectively.

Function `memcpy` copies `n` characters from memory area `source` to `dest`. It returns `dest`.

Function `memset` sets the first `n` characters in memory area `dest` to the value of character `c`. It returns `dest`.

Warning

Overlapping moves yield unexpected results.

See also

`string`, `BlockMove` in the *Apple IIGS Toolbox Reference*

onexit—install a function to be executed at program termination

Synopsis

```
int onexit(func);
  void (*func)();
```

Description

Function `onexit` installs the `exit` function pointed to by `func` by adding it to a list. The list is initially empty. A list entry is added whenever `onexit` is called. Function `exit` calls the functions in the list in the reverse of the order in which they were added.

Programs that use the buffered I/O portions of the Standard I/O Package (including the predefined streams `stdin`, `stdout`, and `stderr`) need to flush all open buffers before the program terminates. To ensure that this is done, the Standard I/O Package adds its cleanup function to the list the first time that it allocates a buffer. Each function in the list is called with a single argument of type `int` either at program termination or when `exit` is called. This argument is the program's status value (0 for normal execution; nonzero for errors). The function can use this value or ignore it.

The number of user-supplied exit functions is limited to six, including the one used by the Standard I/O Package.

Diagnostics

The function returns a nonzero value if the installation fails.

Note

A call to `_exit` circumvents user exit procedures installed by `onexit`.

Warning

The behavior of a function is undefined if it is installed more than once.

See also

`exit`, `stdio`

open—open for reading or writing

Synopsis

```
#include <FCNTL.H>
int open(filename, oflag)
  char *filename;
  int oflag;
```

Description

Parameter `filename` is a filename or pseudo-filename (such as `.STDIN`, `.STDOUT`, `.STDERR`, `.CONSOLE`, or `.NULL`); it cannot be the pseudo-filename `.printer`, a device name (such as `.D2`), or the double period (`..`).

Function open opens a file descriptor for the named file and sets the file-status flags according to the value of `oflag`. The value of `oflag` is constructed by OR-ing flag settings; for example,

```
fildes = open("MyFile", O_WRONLY|O_CREAT|O_TRUNC);
```

To construct `oflag`, first select one of the following access modes:

`O_RDONLY`	Open for reading only
`O_WRONLY`	Open for writing only
`O_RDWR`	Open for reading and writing

Then optionally add one or more of these modifiers:

`O_APPEND`	The file pointer is set to the end of file before each write operation.
`O_CREAT`	If the file does not exist, it is created.
`O_TRUNC`	If the file exists, its length is truncated to 0; the mode is unchanged.

The following setting is valid only if `O_CREAT` is also specified:

`O_EXCL`	Function open fails if the file exists.

When a file is written to a device, normally certain characters are translated to match the needs of the device or the expectations of ProDOS for a normal text file (such as translating `\n` to CR rather than LF). The following flag suppresses such translation.

`O_BINARY`	The file is read or written verbatim, suppressing the device driver's conversions.

Upon successful completion, a nonnegative integer (the file descriptor) is returned. The file pointer used to mark the current position within the file is set to the beginning of the file.

The named file is opened unless one or more of the following is true:

`O_CREAT` is not set and the named file does not exist. [`ENOENT`]
More than about 30 file descriptors are currently open. The actual limit varies according to run-time conditions. [`ENFILE`]
`O_CREAT` and `O_EXCL` are set, and the named file exists. [`EEXIST`]

Return value

Upon successful completion, this function returns a nonnegative integer (the file descriptor). Otherwise, it returns a value of –1 and sets `errno` to indicate the error.

See also

`close`, `creat`, `lseek`, `read`, `write`

printf—print formatted output

Synopsis

```
#include <STDIO.H>

int printf(format [ , arg ] ... )
  char *format;
int fprintf(stream, format [ , arg ] ... )
  FILE *stream;
  char *format;
int sprintf(str, format [ , arg ] ... )
  char *str, *format;
```

Description

Function `printf` places formatted output on the standard output stream `stdout`. Function `fprintf` places formatted output on the named output stream `stream`. Function `sprintf` places formatted output, followed by the null character (`\0`), into the character array pointed to by `str` (you must ensure that enough room is available). Each function returns the number of characters transmitted (not including the `\0` in the case of `sprintf`) or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its `arg` parameters under control of the `format` parameter. The `format` parameter is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching zero or more `arg` parameters. The behavior of the function is undefined if there are insufficient `arg` parameters for the format. If the format is exhausted while `arg` parameters remain, the extra `arg` parameters are ignored.

Each conversion specification is introduced by the character `%`. After `%`, the following appear in sequence:

1. Zero or more flag characters, which modify the meaning of the conversion specification.
2. An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, the value will be padded to the field width on the left (default) or right (if the left-adjustment flag has been given): see the discussion of flag specification that follows.
3. A precision that gives the minimum number of digits to appear for the `d`, `o`, `u`, `x`, and `X` conversions; the number of digits to appear after the decimal point for the `e`, `E`, and `f` conversions; the maximum number of significant digits for the `g` and `G` conversions; or the maximum number of characters to be printed from a string in the `s` conversion. The format of the precision is a period (`.`) followed by a decimal digit string; a null digit string is treated as zero.
4. An optional `l` specifying that a following `d`, `o`, `u`, `x`, or `X` conversion character applies to an `arg` parameter of type `long`.
5. A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (`*`) instead of a digit string. In this case, an integer `arg` parameter supplies the field width or precision. The `arg` parameter that is actually converted is not fetched until the conversion letter is seen; therefore, the `arg` parameters specifying field width or precision must appear immediately before the `arg` parameter (if any) to be converted.

These are the flag characters and their meanings:

-	The result of the conversion will be left justified within the field.
+	The result of a signed conversion always begins with a sign (+ or -).
`blank`	If the first character of a signed conversion is not a sign, a space will be prefixed to the result. This prefix implies that if the `blank` and + flags both appear, the `blank` flag will be ignored.
#	The value is to be converted to an alternate form. For `c`, `d`, `s`, and `u` conversions, the flag has no effect. For `o` conversion, the flag increases the precision to force the first digit of the result to be 0. For `x` (`X`) conversion, a nonzero result will have `0x` (`0X`) prefixed to it. For `e`, `E`, `f`, `g`, and `G` conversions, the result will always contain a decimal point, even if no digits follow the decimal point. (Normally, a decimal point appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros in the fractional part will not be removed from the result (as they normally are).

Here are the conversion characters and their meanings:

`d, o, u, x, X`	The integer `arg` parameter is converted to signed decimal (`d`), unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` and `X`). The letters `abcdef` are used for `x` conversion, and the letters `ABCDEF` are used for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of 0 is a null string.
`f`	The `float`, `double`, `comp`, or `extended` `arg` parameter is converted to decimal notation in the form "[-]*ddd.ddd*", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, it is assumed to be 6; if the precision is explicitly 0, no decimal point appears. Infinities are printed in the form "[-]`INF`", and NaNs are printed in the form "[-]`NAN(`*ddd*`)`", where *ddd* is a code indicating why the result is not a number.
`e, E`	The `float`, `double`, `comp`, or `extended` `arg` parameter is converted to decimal notation in the form "[-]`d.`*ddd*`e±`*dd*", where there is one digit before the decimal point, and the number of digits after the decimal point is equal to the precision. When the precision is missing, it is assumed to be 6; if the precision is 0, no decimal point appears. The `E` format code produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. Infinities are printed as `INF`, and NaNs are printed in the form "[-]`NAN(`*ddd*`)`", where *ddd* is a code indicating why the result is not a number.

g, G	The `float`, `double`, `comp`, or `extended` `arg` parameter is printed in style `f` or `e` (or in style `f` or `E` in the case of a `G` format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style `e` is used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeros are removed from the result. A decimal point appears only if it is followed by a digit.
c	The `char` `arg` parameter is printed.
s	The `arg` parameter is taken to be a string (character pointer), and characters from the string are printed until a null character (`\0`) is encountered or until the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, with the result that all characters up to the first null character are printed. If the string pointer `arg` parameter has the value zero, the result is undefined; a zero `arg` parameter yields undefined results.
p	The `arg` parameter is taken to be a Pascal string, which begins with a character specifying its length and does not end with a null character (`\0`).
%	The `%` character is printed; no parameter is converted.

In no case does a nonexistent or small field width cause truncation of a field. If the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by `printf` and `fprintf` are printed as if `putc` had been called.

Examples

To print a date and time in the form "Sunday, July 3, 10:02", where `weekday` and `month` are pointers to null-terminated strings, use

```
printf("%s, %s %d, %.2d:%.2d", weekday, month, day, hour, min);
```

To print pi to five decimal places, use

```
printf("pi = %.5f", pi());
```

Note

Calling `sprintf` causes other Standard I/O functions to be loaded, even though `sprintf` doesn't perform any I/O operations.

See also

`ecvt`, `putc`, `scanf`, `stdio`

"Conversions Between Decimal Formats" in Chapter I-401 in the *Apple Numerics Manual*.

putc—put character or word on a stream

Synopsis

```
#include <STDIO.H>

int putc(c, stream)
  char c;
  FILE *stream;
int putchar(c)
  char c;
int fputc(c, stream)
  char c;
  FILE *stream;
int putw(w, stream)
  int w;
  FILE *stream;
```

Description

Macro `putc` writes the character `c` to the output stream at the current position of the file pointer. Macro `putchar(c)` is equivalent to

```
putc(c, stdout)
```

Function `fputc` behaves like macro `putc`. Function `fputc` runs more slowly than macro `putc` but takes less space per invocation.

Function `putw` writes an `int` (that is, 2 bytes) to the output stream at the current position of the file pointer. This function neither assumes nor causes special alignment in the file.

For information about output files buffering, see `stdio`.

Return values

When `putc`, `putchar`, or `fputc` succeeds, it returns the value it has written. When one of these fails, it returns the constant `EOF` (–1). (These functions fail if the file stream is not open for writing, or if the output file cannot be grown.)

When `putw` succeeds, it returns 0; when it fails, it returns a nonzero value.

Note

Because `putc` is implemented as a macro, it treats a `stream` parameter with side effects incorrectly. In particular,

```
putc(c, *f++)
```

produces unexpected results. Instead, use

```
fputc(c, *f++)
```

See also

`fclose`, `ferror`, `fopen`, `fread`, `getc`, `printf`, `puts`, `setbuf`, `stdio`

puts—write a string to a stream

Synopsis

```
#include <STDIO.H>

int puts(str)
  char *str;
int fputs(str, stream)
  char *str;
  FILE *stream;
```

Description

Function `puts` writes the null-terminated string pointed to by `str`, followed by a newline character, to the standard output stream `stdout`.

Function `fputs` writes the null-terminated string pointed to by `str` to the named output stream `stream`.

Neither function writes the terminating null character.

Return value

Both routines return either the number of characters written, or return `EOF` (–1) if a write error occurs.

Note

Function `puts` appends a newline character, while `fputs` does not.

See also

`ferror`, `fopen`, `fread`, `printf`, `putc`, `stdio`

qsort—quicker sort

Synopsis

```
void qsort(base, nelem, elsize, compar)
  char *base;
  unsigned int nelem, elsize;
  int (*compar)();
```

Description

Function `qsort` is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

Parameter `base` points to the element at the base of the table. Parameter `nelem` is the number of elements in the table. Parameter `elsize` is the size of an element in the table; it can be specified as `sizeof(*base)`.

Parameter `compar` is a pointer to a comparison function that you supply. Function `qsort` calls your comparison function with pointers to two elements being compared. Here is a sample declaration for your comparison function:

```
int myCompare(elem1, elem2)
      char *elem1, *elem2;
```

Your comparison function supplies the result of the comparison to `qsort` by returning one of the following integer values:

Result	Meaning
<0	The first parameter is less than the second parameter
0	The first parameter is equal to the second parameter
>0	The first parameter is greater than the second parameter

Note

Parameter `base`, the pointer to the base of the table, should be of the pointer-to-element type and cast to `(char *)`.

rand—a simple random-number generator

Synopsis

```
int rand()
void srand(seed)
  unsigned seed;
```

Description

Function `rand` uses a multiplicative congruential random-number generator with a period of 2^{32} that returns successive pseudorandom numbers in the range from 0 to $2^{15}-1$.

Function `srand` can be called at any time to reset the random-number generator to a specific seed. The generator is initially seeded with a value of 1. Identical seeds produce identical sequences of pseudorandom numbers.

See also

"Random Number Generator" in Chapter I-10 of the *Apple Numerics Manual*

read—read from file

Synopsis

```
int read(fildes, buf, nbyte)
  int fildes;
  char *buf;
  unsigned nbyte;
```

Description

File descriptor `fildes` is obtained from a call to open, `creat`, dup, or `fcntl`.

Function `read` transfers up to `nbyte` bytes from the file associated with `fildes` into the buffer pointed to by `buf`.

On devices capable of seeking, `read` starts reading at the current position of the file pointer associated with `fildes`. Upon returning from `read`, the file pointer is incremented by the number of bytes actually read.

Nonseeking devices always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, `read` returns the number of bytes actually read and placed in the buffer; this number may be less than `nbyte` if the number of bytes left in the file is less than `nbyte` bytes. A value of 0 is returned when the end of file has been reached; a vlaue of –1 if a read error occurred.

Function `read` fails if `fildes` is not a valid file descriptor associated with a file open for reading. [`EBADF`]

File descriptor 0 is opened by the APW Shell as the standard input file.

Return value

Upon successful completion, a nonnegative integer (the function `read`) returns the number of bytes actually read. Otherwise, it returns –1 and sets `errno` to indicate the error.

See also

`creat`, `open`

scanf—convert formatted input

Synopsis

```
#include <STDIO.H>

int scanf(format [ , pointer ] ... )
  char *format;
int fscanf(stream, format [ , pointer ] ... )
  FILE *stream;
  char *format;
int sscanf(str, format [ , pointer ] ... )
  char *str, *format;
```

Description

Function `scanf` reads characters from the standard input stream `stdin`. Function `fscanf` reads characters from the named input stream `stream`. Function `sscanf` reads characters from the character string `str`. Each function converts the input according to a control string (`format`), and stores the results according to a set of `pointer` parameters that indicate where the converted output should be stored.

Parameter `format`, the control string, contains specifications that control the interpretation of input sequences. The format consists of characters to be matched in the input stream, conversion specifications that start with the character `%`, or both. The control string may contain the following:

- white-space characters (spaces and tabs) that cause input to be read up to the next non-white-space character, except as described here
- a character (any except `%`) that must match the next character of the input stream; to match a `%` character in the input stream, use `%%`
- conversion specifications beginning with the character `%` and followed by an optional assignment-suppression character `*`; an optional numeric maximum field width; an optional `l`, `m`, `n`, or `h` indicating the size of the receiving parameter; and a conversion code

An input field is defined relative to its conversion specification. The input field ends when the first character inappropriate for conversion is encountered or when the specified field width is exhausted. After conversion, the input pointer points to the inappropriate character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding parameter, which is a pointer to a basic C type, such as `int` or `float`.

Assignment can be suppressed by preceding a format character with the character `*`. In assignment suppression, an input field is skipped: the field is read and converted, but not assigned. Therefore, `pointer` should be omitted when assignment of the corresponding input field is suppressed.

The **format character** dictates the interpretation of the input field. The following format characters are legal in a conversion specification, after `%`:

`%`	A single `%` is expected in the input at this point. Assignment is not performed.
`d`	A decimal integer is expected. The corresponding parameter should be an integer pointer.
`u`	An unsigned decimal integer is expected. The corresponding parameter should be an unsigned integer pointer.

o	An octal integer is expected. The corresponding parameter should be an integer pointer.
x	A hexadecimal integer is expected. The corresponding parameter should be an integer pointer. The conversion characters d, u, o, and x may be preceded by l or h to indicate that a pointer to long or short, rather than int, is in the parameter list. The h is ignored in this implementation because int and short are both 16 bits.
e, f, g	A floating-point number is expected. The next field is converted accordingly and stored through the corresponding parameter, which should be a pointer to float, double, comp, or extended, depending on the size specification. The input format for floating-point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of E or e followed by an optionally signed integer. In addition, infinity is represented by the string "INF", and NaNs are represented by the string "NAN", optionally followed by parentheses that may contain a string of digits (the NaN code). Case is ignored in the infinity and NaN strings. The conversion characters e, f, and g may be preceded by l, m, or n to indicate that a pointer to double, comp, or extended, rather than float, is in the parameter list.
s	A character string is expected. The corresponding parameter should be a character pointer to an array of characters large enough to accept the string; a terminating null character (\0) is added automatically. The input field is terminated by a white-space character (space or tab), or when the number of characters specified by the maximum field width has been read.
p	A character string is expected. The next field is converted to a Pascal-format string—that is, a character specifying the length of the string followed by the string itself. The corresponding parameter should be a character pointer to an array of characters large enough to accept the string; a terminating null character (\0) is added automatically. The input field is terminated by a white-space character (space or tab), or when the number of characters specified by the maximum field width has been read.
c	A character is expected; the corresponding parameter should be a character pointer. The normal skip over white space is suppressed in this case; use %1s to read the next non-white-space character. If a field width is given, the corresponding parameter should refer to a character array; the indicated number of characters is read.
[	The left bracket is followed by a set of characters called the **scanset** and a terminating right bracket. The input field is the maximal sequence of input characters consisting entirely of characters in the scanset. When reading the input field, the normal skip over leading white space is suppressed. The corresponding pointer parameter must point to a character array large enough to hold the input field and the terminating null character (\0), which will be added automatically.

^ When appearing as the first character in the scanset, the circumflex serves as a complement operator, and redefines the scanset as the set of all characters not contained in the remainder of the scanset string.

] The right bracket ends the scanset. To be included as an element of the scanset, the right bracket must appear as the first character (possibly preceded by a circumflex) of the scanset. Otherwise, it will be interpreted syntactically as the closing bracket.

A range of characters may be represented by the construct *first-last;* thus, the scanset `[0123456789]` may be expressed `[0-9]`. To use this convention, `first` must be less than or equal to `last` in the ASCII collating sequence. Otherwise, the minus (-) will stand for itself in the scanset. The minus will also stand for itself whenever it is the first or the last character in the scanset. At least one character must match for the conversion to be considered successful.

Conversion terminates when the end of file or the end of the control string is reached, or when an input character doesn't match the control string. In the last case, the unmatched character is left unread in the input stream.

Examples

Here are some ways that the `scanf` function can be used:

- The call

```
int i;
float x;
char name[50];
scanf("%d%f%s", &i, &x, name);
```

with input

```
25 54.32E-1 reed
```

will assign the value 25 to `i` and the value 5.432 to `x`; `name` will contain "`reed\0`".

- The call

```
int i;
extended x;
char name[50];
scanf("%2d%nf%*d %[0-9]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to `i` and 789.0 to `x`, skip 0123, and place the string "`56\0`" in `name`. The next call to `getchar` will return "`a`".

- The call

```
int i;
scanf("answer1=%d", &i);
```

with input

```
answer1=51 answer2=45
```

will assign the value 51 to `i` because "`answer1`" is matched explicitly in the input stream. The input pointer will be left at the space before "`answer2`".

Return value Functions `scanf`, `fscanf`, and `sscanf` return the number of successfully matched and assigned input items. This number can be zero when an early mismatch between an input character and the control string occurs. If the input ends before the first mismatch or conversion, `EOF` is returned.

These functions return `EOF` when input ends, and a short count for missing or illegal data items.

Note Trailing white space is left unread unless it is matched in the control string.

The success of literal matches and suppressed assignments cannot be determined.

Warning The `pointer` parameters in these functions must be addresses: for example, `&i`. Be sure not to pass `i` rather than its address.

See also `atof`, `getc`, `printf`, `stdio`, `strtol`

"Conversions Between Decimal Formats" in Chapter I-4 of the *Apple Numerics Manual*

setbuf—assign buffering to a stream

Synopsis

```
#include <STDIO.H>

void setbuf(stream, buf)
  FILE *stream;
  char *buf;
int setvbuf(stream, buf, type, size)
  FILE *stream;
  char *buf;
  int type;
  int size;
```

Description

A buffer is normally allocated by the Standard C Library at the time of the first `getc` or `putc` operation on a file. If you prefer to provide your own buffer, you can call `setbuf` or `setvbuf` after a stream has been associated with an open file but before it is read or written. Functions `setbuf` and `setvbuf` let you provide your own buffering for a file stream. Function `setvbuf` is a more flexible extension of `setbuf`.

Function `setbuf` causes the character array pointed to by `buf` to be used instead of an automatically allocated buffer. `BUFSIZ`, a constant defined in the `<StdIO.h>` header file, specifies the size of the `buf` array as

```
char buf[BUFSIZ];
```

If `buf` is `NULL`, input and output are unbuffered.

Function `setvbuf` lets you specify two parameters in addition to those required by `setbuf`: `size` and `type`. Parameter `size` specifies the size in bytes of the array to be used; the standard I/O functions work most efficiently when `size` is a multiple of `BUFSIZ`. If the buffer pointer `buf` is `NULL`, a buffer of `size` bytes is allocated from the system. If `buf` is not `NULL`, `buf` is assigned to the `FILE` variable's buffer-pointer parameter. If `size` is not zero, `size` is assigned to the `FILE` variable's size parameter. The value of `type` determines how `stream` is buffered by `setvbuf`, as follows:

Value of type	Description
`_IOFBF`	Causes input and output to be file buffered.
`_IOLBF`	Causes output to be line buffered. The buffer is flushed either when a newline character is written or when the buffer is full.
`_IONBF`	Causes input and output to be unbuffered. Parameters `buf` and `size` are ignored.

The following function calls are equivalent when `buf` is not `NULL`:

```
setbuf(stream, buf);
setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

The following function calls are equivalent when `buf` is `NULL`:

```
setbuf(stream, NULL);
setvbuf(stream, NULL, _IONBF, 0);
```

Diagnostics Function `setvbuf` returns nonzero if an invalid value is given for `type`.

Note The buffer must have a life at least as long as that of the open stream. Be sure to close the stream before the buffer is deallocated. If you allocate buffer space as an automatic variable in a code block, be sure to close the stream in the same block.

If `buf` is `NULL` and the system cannot allocate `size` bytes, a smaller buffer will be allocated.

See also `fopen`, `getc`, `malloc`, `putc`, `stdio`

setjmp—nonlocal transfer of control

Synopsis

```
#include <SETJMP.H>

int setjmp(env)
  jmp_buf env;
void longjmp(env, val)
  jmp_buf env;
  int val;
```

Description

These functions let you escape from an error or interrupt encountered in a low-level subroutine of your program.

Function `setjmp` saves its stack environment in `env` for later use by `longjmp`. It returns the value 0.

Function `longjmp` restores the environment saved by the last call of `setjmp` with the corresponding `env` environment. After a call to `longjmp`, the program continues as if the preceding call to `setjmp` had returned the value `val`.

Function `longjmp` cannot cause `setjmp` to return the value 0. If `longjmp` is invoked with a second parameter of 0, `setjmp` returns 1. Data values will be those in effect at the time `longjmp` was called.

Warning

If `longjmp` is called without a previous call to `setjmp`, or if the function that contained the `setjmp` has already returned, results are unpredictable.

sinh—hyperbolic functions

Synopsis

```
#include <MATH.H>

extended sinh(x)
  extended x;
extended cosh(x)
  extended x;
extended tanh(x)
  extended x;
```

Description

Functions `sinh`, `cosh`, and `tanh` return, respectively, the hyberbolic sine, cosine, and tangent of their parameter.

Diagnostics

Functions `sinh`, `cosh`, and `tanh` honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

"Exception Flags and Halts" in Chapter I-8, and Appendix A of the *Apple Numerics Manual*

stdio—standard buffered input/output package

Synopsis

```
#include <STDIO.H>

FILE *stdin, *stdout, *stderr;
```

Description

The Standard I/O Package constitutes an efficient user-level I/O buffering scheme. The inline macros `getc` and `putc` handle characters quickly. Macros `getchar` and `putchar`, and the higher-level routines `fgetc`, `fgets`, `fprintf`, `fputc`, `fputs`, `fread`, `fscanf`, `fwrite`, `gets`, `getw`, `printf`, `puts`, `putw`, and `scanf` all use `getc` and `putc`. Calls to these macros and functions can be freely intermixed.

The constants and the following functions are implemented as macros: `getc`, `getchar`, `putc`, `putchar`, `feof`, `ferror`, `clearerr`, and `fileno`. Redeclaration of these names should be avoided.

Any program that uses the Standard I/O Package must include the `StdIO.h` header file of macro definitions. The functions, macros, and constants used in the Standard I/O package are declared in the header file and need no further declaration.

A *stream* is a file with associated buffering and is declared to be a pointer to a `FILE` variable. Functions `fopen`, `freopen`, and `fdopen` return this pointer. The information in the `FILE` variable includes

- the file access—read or write
- the file descriptor as returned by `open`, `creat`, `dup`, or `fcntl`
- the buffer size and location
- the buffer style (unbuffered, line-buffered, or file-buffered)

Standard I/O buffering

Output streams, with the exception of the standard error stream `stderr`, are by default file buffered if the output refers to a file. File `stderr` is by default line buffered. When an output stream is *unbuffered,* it is queued for writing on the destination file or window as soon as written; when it is *file buffered,* many characters are saved up and written as a block; when it is *line buffered,* each line of output is queued for writing as soon as the line is completed (that is, as soon as a newline character is written). Function `setvbuf` may be used to change the stream's buffering strategy.

Normally, there are three open streams with constant pointers declared in the `<STDIO.H>` header file and associated with the standard open files:

FILE variable	Files	Description	Buffer style
`stdin`	0	Standard input file	Line buffered
`stdout`	1	Standard output file	File buffered
`stderr`	2	Standard error file	Line buffered

Buffer initialization

The FILE variable returned by fopen, freopen, or fdopen has an initial buffer size of 0 and a NULL buffer pointer. The buffer size is set and the buffer allocated by a call to setbuf, setvbuf, or the first I/O operation on the stream, whichever comes first. Buffer initialization is performed using the following algorithm:

1. If _IONBF (no buffering) was set by a call to setvbuf, initialization steps 2 and 3, that follow, are skipped. The buffer size remains 0, and the buffer pointer remains NULL.
2. The access-mode word for _IOLBF (line buffering) is checked. This bit is usually set only in the predefined files stderr and stdin, but a call to setvbuf can set it for any file. If line buffering is set, the buffer size is set to _LBUFSIZ (100). If line buffering is not set, ioctl is called with an FIOBUFSIZE request, and the buffer size is set to the returned value or to BUFSIZ (1024) if no value is returned.
3. If the buffer pointer is NULL, a request is made for a buffer whose size was determined in step 2; the buffer pointer is set to point to the newly allocated buffer. If the requested size cannot be allocated, attempts are made to allocate first BUFSIZ and then _LBUFSIZ if one of these is smaller than the requested size. If all requests fail, the buffer pointer remains NULL, and the _IONBF (no buffering) bit is set.
4. Function ioctl is called with an FIOINTERACTIVE request; if it returns true, the _IOSYNC bit is set in the access-mode word. This step is done for all FILE variables, regardless of their buffering style and size. (The _IOSYNC bit is described in the following section.)

The setvbuf function lets you specify values for buffer size, buffer pointer, and access mode word other than the default values of 0, NULL, and 0, respectively. The setvbuf function must be called before the first I/O operation occurs, so that the buffer-initialization procedure just described receives the values you specify instead of the default values.

Buffered I/O

On each write request, the bytes are transferred to the buffer, and an internal counter is set to account for the number of bytes in the buffer. If _IOLBF is set and a newline character is encountered while bytes are being transferred to the buffer, the buffer is flushed (written immediately) and the transfer continues at the beginning of the buffer. This process continues until the write-request count is satisfied or a write error occurs.

On each read request, the _IOSYNC bit in the access-mode word is checked. If _IOSYNC is on, all current FILE variables that have _IOSYNC on and that are open for writing are flushed. In other words, a read operation from an interactive FILE variable flushes all interactive output files before reading is performed. This process ensures that any prompts, screen output, or other visual feedback is displayed before the read operation is initiated. Then if the internal counter is 0, an entire buffer is read into memory if possible. (For the console device, less than a buffer's worth is likely to be read.) The bytes required to satisfy the read request are transferred; the device is asked for more bytes if necessary, and an internal pointer is advanced if any bytes remain unread.

When the Standard I/O Package is used, Standard I/O cleanup is performed just before the application terminates. Any normal return including a call to `exit` causes Standard I/O cleanup, which consists of a call to `fclose` for every open `FILE` stream.

Note

Do not use a file descriptor (0, 1, or 2) where a `FILE` variable (`stdin`, `stdout`, or `stderr`) is required.

File `StdIO.h` includes definitions other than those just described, but their use is not recommended.

Invalid stream pointers can cause serious errors, including program termination. Individual function descriptions describe the possible error conditions.

Diagnostics

Most integer functions that deal with streams return the integer constant `EOF` (–1) when the end of file is reached or when an error occurs. See the descriptions of the individual functions for details.

See Also

`open`, `close`, `lseek`, `read`, `write`, `fclose`, `ferror`, `fopen`, `fread`, `fseek`, `getc`, `gets`, `printf`, `putc`, `puts`, `scanf`, `setbuf`, `ungetc`

string—string operations

Synopsis

```
#include <STRING.H>
#include <STRINGS.H>

char *strcat(destStr, srcStr)
  char *destStr, *srcStr;
char *strncat(destStr, srcStr, n)
  char *destStr, *srcStr;
  int n;
int strcmp(str1, str2)
  char *str1, *str2;
int strncmp(str1, str2, n)
  char *str1, *str2;
  int n;
char *strcpy(destStr, srcStr)
  char *destStr, *srcStr;
char *strncpy(destStr, srcStr, n)
  char *destStr, *srcStr;
  int n;
int strlen(str)
  char *str;
char *strchr(str, c)
  char *str, c;
char *strrchr(str, c)
  char *str, c;
char *strpbrk(srcStr, findChars)
  char *srcStr, *findChars;
int strspn(srcStr, spanChars)
  char *srcStr, *spanChars;
int strcspn(srcStr, skipChars)
  char *srcStr, *skipChars;
char *strtok(destStr, tokenStr)
  char *destStr, *tokenStr;
char *c2pstr(ptr)
  char *ptr
char *p2cstr(ptr)
  char *ptr
```

Description

The string parameters (`srcStr`, `destStr`, and so forth) and `s` point to arrays of characters terminated by a null character. Functions `strcat`, `strncat`, `strcpy`, and `strncpy` all alter `destStr`. These functions do not check for overflow of the array pointed to by `destStr`.

Function `strcat` appends a copy of string `srcStr` to the end of string `destStr`. Function `strncat` appends at most `n` characters. Each function returns a pointer to the null-terminated result.

Function `strcmp` performs a comparison of its parameters according to the ASCII collating sequence and returns an integer less than, equal to, or greater than 0 when `str1` is less than, equal to, or greater than `str2`, respectively. Function `strncmp` makes the same comparison but looks at a maximum of `n` characters.

Function `strcpy` copies string `srcStr` to string `destStr`, stopping after the null character has been copied. Function `strncpy` copies exactly `n` characters, truncating `srcStr` or adding null characters to `destStr` if necessary. The result is not terminated with a null character if the length of `srcStr` is `n` or more. Each function returns `destStr`.

Function `strlen` returns the number of characters in `str`, not including the terminating null character.

Functions `strchr` and `strrchr` both return a pointer to the first and last occurrence, respectively, of character `c` in string `str`; they return `NULL` if `c` does not occur in the string. The null character terminating a string is considered to be part of the string. In previous versions of the Standard C Library, `strchr` was known as `index` and `strrchr` was known as `rindex`.

Function `strpbrk` returns a pointer to the first occurrence in string `srcStr` of any character from string `findChars`, or it returns `NULL` if no character from `findChars` exists in `srcStr`.

Function `strspn` returns the length of the initial segment of string `srcStr` that consists entirely of characters from string `spanChars`.

Function `strcspn` returns the length of the initial segment of string `srcStr` that consists entirely of characters not from string `skipChars`.

Function `strtok` considers the string `destStr` as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `tokenStr`. The first call (with pointer `destStr` specified) returns a pointer to the first character of the first token and writes a null character into `destStr` immediately following the returned token. The function keeps track of its position in the string between calls. Subsequent calls for the same string must be made with `NULL` as the first parameter. The separator string `tokenStr` may be different from call to call. When no token remains in `destStr`, `NULL` is returned.

Function `c2pstr` converts in place a C-style string to a Pascal-style string. The function receives a pointer to the string to be converted, and returns a pointer to the converted string.

Function `p2cstr` converts in place a Pascal-style string to a C-style string. The function receives a pointer to the string to be converted, and returns a pointer to the converted string.

Warning

Overlapping moves yield unexpected results.

Functions `strcmp` and `strncmp` use signed arithmetic when comparing their parameters. The sign of the result will be incorrect for characters with values greater than `0x7F` in the Apple IIGS extended character set.

See also

`memory`

`BlockMove`, `EqualString` in the *Apple IIGS Toolbox Reference*

strtol—convert a string to a long

Synopsis

```
#include <STDLIB.H >

long strtol(str, ptr, base)
  char *str;
  char **ptr;
  int base;
```

Description

Function `strtol` returns a `long` containing the value represented by the character string `str`. The string is scanned up to the first character inconsistent with the base (decimal, hexadecimal, or octal). Leading white-space characters are ignored.

If the value of `ptr` is not `NULL`, a pointer to the character terminating the scan is returned in `*ptr`. If no integer can be formed, `*ptr` is set to `str`, and `0` is returned.

If `base` is `0`, the base is determined from the string. If the first character after an optional leading sign is not `0`, decimal conversion is performed, and if the `0` is followed by `x` or `X`, hexadecimal conversion is performed; otherwise, octal conversion is performed.

The function call `atol(str)` is equivalent to

```
strtol(str, (char **)NULL, 10)
```

The function call `atoi(str)` is equivalent to

```
(int) strtol(str, (char **)NULL, 10)
```

Note

Overflow conditions are ignored.

Apple base conventions (`$` for hexadecimal, `%` for binary) are not supported.

See also

`atof`, `atoi`, `scanf`

trig—trigonometric functions

Synopsis

```
#include <MATH.H>

extended sin(x)
  extended x;
extended cos(x)
  extended x;
extended tan(x)
  extended x;
extended asin(x)
  extended x;
extended acos(x)
  extended x;
extended atan(x)
  extended x;
extended atan2(y, x)
  extended y, x;
```

Description

Functions `sin`, `cos`, and `tan` return, respectively, the sine, cosine, and tangent of their argument, which is in radians.

Function `asin` returns the arcsine of `x`, in the range $-\pi/2$ to $\pi/2$.

Function `acos` returns the arccosine of `x`, in the range 0 to π.

Function `atan` returns the arctangent of `x`, in the range $-\pi/2$ to $\pi/2$.

Function `atan2` returns the arctangent of `y/x`, in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value.

For special cases, these functions return a NaN or infinity, as appropriate.

Diagnostics

These functions honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

Note

Functions `sin`, `cos`, and `tan` have periods based on the nearest extended-precision representation of mathematical π. Hence, these functions diverge from their mathematical counterparts as their argument gets further from zero.

See also

"Trigonometric Functions" in Chapter I-10 of the *Apple Numerics Manual*

ungetc—push a character back into the input stream

Synopsis

```
#include <STDIO.H>
int ungetc(c, stream)
  char c;
  FILE *stream;
```

Description

Function ungetc inserts the character c (which typically was returned by the last getc call) into the buffer associated with an input stream. The stream must be file-buffered or line-buffered; it cannot be unbuffered. The inserted character, c, will be returned by the next getc call on that stream. Function ungetc returns c and leaves the file corresponding to stream unchanged.

Only one character of pushback is allowed, provided something has been read from the stream and the stream is not unbuffered.

If c equals EOF, ungetc does nothing to the buffer and returns EOF.

Function ungetc will not clear an end-of-file condition.

Function fseek or rewind undoes the effect of ungetc if the new location is not within the same buffer.

Diagnostics

For ungetc to perform correctly, a read operation (such as getc) must have been performed before the call to the ungetc function. Function ungetc returns EOF if it can't insert the character.

Note

Function ungetc does not work on unbuffered streams.

See also

fseek, getc, setbuf, stdio

unlink—delete a named file

Synopsis

```
int unlink(fileName)
  char *fileName;
```

Description

Function `unlink` deletes the named file. The function fails if the named file is open.

A call to `unlink` is equivalent to

```
faccess(fileName, F_DELETE)
```

Diagnostics

Upon successful completion, this function returns a value of 0. Otherwise, it returns a value of –1 and sets `errno` to indicate the error.

See also

`faccess`

write—write on a file

Synopsis

```
int write(fildes, buf, nbyte)
  int fildes;
  char *buf;
  unsigned nbyte;
```

Description

File descriptor `fildes` is obtained from an `open`, `creat`, `dup`, or `fcntl` call.

Function `write` attempts to write `nbyte` bytes from the buffer pointed to by `buf` to the file associated with `fildes`. Internal limitations may cause `write` to write fewer bytes than requested; the number of bytes actually written is indicated by the return value. Several calls to `write` may therefore be necessary to write the contents of `buf`.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from `write`, the file pointer is incremented by the number of bytes actually written.

On nonseeking devices, writing starts at the current position. The value of a file pointer associated with such a device is undefined.

If the `O_APPEND` file-status flag that is set in `open` is on, the file pointer is set to the end of file before each write operation.

The file pointer remains unchanged, and `write` fails if `fildes` is not a valid file descriptor open for writing. [`EBADF`]

If you try to write more bytes than there is room for on the device, `write` writes as many bytes as possible. For example, if `nbyte` is 512 and there is room for 20 bytes more on the device, `write` writes 20 bytes and returns a value of 20. The next attempt to write a nonzero number of bytes will return an error. [`ENOSPC`]

File descriptor 1 is standard output; file descriptor 2 is standard error.

Return value

Upon successful completion, this function returns the number of bytes actually written. Otherwise, it returns –1 and sets `errno` to indicate the error.

See also

`creat`, `lseek`, `open`

Chapter 6

Shell Calls

The Apple IIGS Progammer's Workshop Shell acts as an interface and extension to ProDOS 16. The shell provides several functions not provided by ProDOS 16; these functions are called exactly like ProDOS 16 functions. Every time a program running under the APW Shell issues a ProDOS 16-like call, the shell intercepts the call. If the call is a shell call, the shell interprets it and acts on it; if it is a ProDOS 16 call, the shell passes it on to ProDOS 16. This chapter describes all of the shell's ProDOS 16-like calls, which are referred to here as **shell calls.**

The shell calls are summarized in Table 6-1. The calls are described in the same order.

Table 6-1
Shell calls

Name	Number	Description
GET_LINFO	(0x0101)	Passes parameters from the shell to a program
SET_LINFO	(0x0102)	Passes parameters from a program to the shell
GET_LANG	(0x0103)	Reads the current language number
SET_LANG	(0x0104)	Sets the current language number
ERROR	(0x0105)	Prints error message for a Apple IIGS tool call
SET_VAR	(0x0106)	Sets the value of a shell variable
VERSION	(0x0107)	Returns the version number of the APW Shell
READ_INDEXED	(0x0108)	Reads variable table
INIT_WILDCARD	(0x0109)	Provides a filename that includes a wildcard character to the shell
NEXT_WILDCARD	(0x010A)	Makes the shell find the next filename matching the wildcard filename
GET_VAR	(0x010B)	Reads the value of a shell variable
EXECUTE	(0x010D)	Sends a command or list of commands to the shell command interpreter
DIRECTION	(0x010F)	Tells whether I/O redirection has occurred
REDIRECT	(0x0110)	Sets device and file for I/O redirection
STOP	(0x0113)	Detects a request for an early termination of the program
WRITE_CONSOLE	(0x011A)	Sends output to the console

How to make a shell call

To make a shell call, you should do the following:

- Include the statements

```
#include    <TYPES.H>
#include    <SHELL.H>
```

  in your source text. Your object file will be automatically linked with the library file CLIB.

- Set values in the shell data structures and call the shell routines from your program, following the information given next.

How a program makes a shell call

A C program makes a shell call by calling a function in the file SHELL.C. Most of these calls are simple C function calls: parameters are passed in the normal way.

Two of these, GET_LINFO and SET_LINFO, are called differently. Values and results are passed via a parameter block. To get information from the shell, your program declares and initializes this parameter block, calls GET_LINFO, and then reads results from the block. To send information to the shell, your program writes values into the block, then calls SET_LINFO to send the information. These calls are explained in detail in the section "GET_LINFO and SET_LINFO," that follow.

With the exception of EXECUTE, all calls expect Pascal-style strings.

Call descriptions

This section lists each shell call, describes its use, and describes the contents of its parameter block. The possible errors returned by a call are listed at the end of each call description. The calls are listed in order of their call numbers.

GET_LINFO (0x0101) and SET_LINFO (0x0102)

```
void GET_LINFO(pb)
       GetLInfoPB *pb;

void SET_LINFO(pb)
       GetLInfoPB *pb;
```

The GET_LINFO function is used by an assembler, compiler, linker, or editor to read the parameters that are passed to it. When you make this call, you declare the parameter block GetLInfoPB; when the APW Shell returns control to your program, you can then read the parameter block to obtain the information you need.

Use the GET_LINFO call to read parameters passed to your assembler, compiler, linker, or editor.

The SET_LINFO function is used by an assembler, compiler, linker, or editor to pass parameters to the APW Shell before returning control to the shell. It can also be used by a shell program under which you are running APW to pass parameters to the APW Shell.

Use the SET_LINFO call when your program is finished before returning control to the shell.

Both of these calls use the following parameter block:

```
GetLInfoPB /* get/set Line Info parameter block */
typedef struct {
   char *sfile;            /* address of source file name */
   char *dfile;            /* address of output file name */
   char *parms;            /* address of parameter list */
   char *istring;          /* address of language-specific input string */
   char merr;              /* maximum error level allowed */
   char merrf;             /* maximum error level found */
   char lops;              /* operations flag */
   char kflag;             /* KEEP flag */
   unsigned long mflags;   /* set of letters selected with '-' */
   unsigned long pflags;   /* set of letters selected with '+' */
   unsigned long org;      /* abs start address of non-reloc load file */
} GetLInfoPB;
```

To call GET_LINFO, first declare the parameter block GetLInfoPB. The GET_LINFO call passes to the shell the pointer, pb, to your parameter block. The shell then writes its results into your parameter block: you can read them from there.

To call SET_LINFO, first declare the parameter block GetLInfoPB, then write your values into that block. The SET_LINFO call passes to the shell the pointer, pb, to your parameter block. The shell then reads your values from the parameter block.

The sfile (source file) field is the address of a buffer containing the filename of the source file; that is, the next file that a compiler or assembler is to process. The filename can be any valid ProDOS 16 filename, and can be a partial or full pathname.

The dfile (destination file) field is the address of a buffer containing the filename of the output file (if any); that is, the file that the compiler or assembler writes to. The filename can be any valid ProDOS 16 filename, and can be a partial or full pathname.

The parms field is the address of a buffer containing the list of names from the NAMES= parameter list in the APW Shell command that called the assembler or compiler. The compiler can remove or modify these names as it processes them, so this list can be different from the one received through the GET_LINFO call..

The istring field is a placeholder for the address of a buffer containing the string of commands passed to the compiler. This command string is not reused by the shell, so it is not necessary to pass it back to the shell with the SET_LINFO call.

The merr field is the maximum error level allowed. If the maximum error level found by the assembler, compiler, or linker is greater than merr, then the shell does not call the next program in the processing sequence. For example, if you use the ASML command to assemble and link a program, but the assembler finds an error level of 8 when merr equals 2, then the linker is not called when the assembly is complete.

The merrf field is the maximum error level found. If merrf is greater than merr, then no further processing is done by the shell. If the high bit of merrf is set, then merrf is considered to be negative; a negative value of merrf indicates a fatal error (normally, all fatal errors are flagged by setting merrf= to 0xFF). In this case, processing terminates immediately and the shell passes control to the APW Editor.

The lops field comprises the operation flags. This field keeps track of the operations that have been performed, and remain to be performed, by the system. The format of this byte is as follows:

Bit:	7	6	5	4	3	2	1	0
Value:	0	0	0	0	0	E	L	C

C = Compile
E = Execute
L = Link

When a bit is set (1), the indicated operation is to be done. When a compiler finishes its operation and returns control to the shell, it clears bit 0 unless a file with another language is appended to the source. When a linker returns control to the shell, it clears bit 1. When you execute the APW Linker by compiling a LinkEd file, the linker clears bits 0 and 1.

The kflag field is the keep flag. This flag indicates what should be done with the output of a compiler, assembler, or linker, as follows:

Kflag value	**Meaning**
0x00	Do not save output.
0x01	Save to an object file with the root filename pointed to by dfile. For example, if the output filename pointed to by dfile is PROG, then the first segment to be executed should be put in PROG or PROG.ROOT, and the remaining segments should be put in PROG.A. For linkers, save to a load file with the name pointed to by dfile (for example, PROG). A compiler or assembler will never set kflag to 0x01, but a shell program calling APW might use this value.
0x02	The .ROOT file has already been created. In this case, the first file created by the next compiler or assembler should end in the .A extension.
0x03	At least one alphabetic suffix has been used. In this case, the compiler or assembler must search the directory for the highest alphabetic suffix that has been used, and then use the next one. For example, if PROG.ROOT, PROG.A, and PROG.B already exist, the compiler should put its output in PROG.C.

When the compiler or assembler passes control back to the shell, it should reset kflag to indicate which object files it has written; for example, if it found only one segment and created a .ROOT file but no .A file, then kflag should be 0x02 in the SET_LINFO call. See "Compilers and Assemblers" in Chapter 6 of the *APW Reference* for more information on object-file naming conventions.

The mflags (minus flags) field passes the flags with a minus sign. This field passes command-line-option flags, such as -L or -C. The first 26 bits of these 4 bytes represent the letters A–Z, arranged with A as the most significant bit of the most significant byte; the bytes are ordered least significant byte first. The bit map is as follows:

```
11000000 11111111 11111111 11111111

YZ       QRSTUVWX IJKLMNOP ABCDEFGH
```

For each flag set with a minus sign in the command, the corresponding bit in this field is set to 1. See the discussions of the `ALINK` and `ASML` commands in Chapter 3 of the *APW Reference* for descriptions of these option flags.

The `pflags` (plus flags) field passes the flags with a plus sign. This field passes command-line-option flags such as `+L` or `+C`. The first 26 bits of these 4 bytes represent the letters A–Z; the bit map for this field is the same as for the `mflags` field. See the discussions of the `ALINK` and `ASML` commands in Chapter 3 of the *APW Reference* for descriptions of these option flags.

The `org` field specifies the absolute start address of a nonrelocatable load file, if one has been specified. This field is only useful in assembly language, and is used only by the linker. C does not use this field.

Possible errors

None

GET_LANG (0x0103)

```
unsigned int GET_LANG()
```

This function reads the current language number. The current language number is set either by the APW Editor when it opens an existing file, or by the user with an APW Shell command. Language numbers are described in "Command Types and the Command Table" in Chapter 3 of the *APW Reference,* and are listed in Appendix B of the *APW Reference.*

Possible errors

None

SET_LANG (0x0104)

```
void SET_LANG(lang)
       unsigned int lang;
```

This function sets the current language number. Language numbers are described in "Command Types and the Command Table" in Chapter 3 and are listed in Appendix B of the *APW Reference.*

The `lang` parameter is the APW language number to which the current APW language should be set. If the language specified is not installed (that is, not listed in the command table), then the "Language not available" error is returned.

Possible errors

0x80	Language not available

ERROR (0x0105)

```
void ERROR(errnum)
       int errnum;
```

When a Apple IIGS tool call returns an error, your program can use this function to print out the name of the tool and the appropriate error message. This function makes it unnecessary for your program to store a complete table of error messages for tool calls. The error number is returned in `_toolErr`.

Possible errors

None

SET_VAR (0x0106)

```
void SET_VAR(varname, value)
       char *varname, *value;
```

This function sets the value of a variable. If the variable has not been previously defined, this function defines it. Variables are described in "Exec Files" in Chapter 3 of the *APW Reference*. Use the `GET_VAR` call to read the current value of a variable and the `READ_INDEXED` call to read a variable table.

The `varname` parameter is a pointer to a buffer in which you place the name of the variable whose value you wish to change. The name is an ASCII string.

The `value` parameter is a pointer to a buffer in which you place the value to which the variable is to be set. The value is an ASCII string.

Possible errors

Errors for Memory Manager calls are described in the *Apple IIGS Toolbox Reference*.

VERSION (0x0107)

```
unsigned long VERSION();
```

This function returns the version of the APW Shell that you are using.

The `VERSION` parameter is a 4-byte ASCII string specifying the version number of the APW Shell that you are using. The initial release returns 10 followed by two space characters (0x3130 0x2020) to indicate version number 1.0.

Possible errors

None

READ_INDEXED (0x0108)

```
void READ_INDEXED(varname, value, index)
       char *varname, *value;
       int index;
```

You can use this function to read the contents of the variable table for the command level at which the call is made. To read the entire contents of the variable table, you must repeat this call, incrementing the index number by 1 each time, until the entire contents have been returned.

The `varname` parameter is a pointer to a 256-byte buffer in which the shell places the name of the next variable in the variable table. The variable name consists of a length byte and a string of ASCII characters. A null string is returned when the index number exceeds the number of variables in the variable table.

The `value` parameter is a pointer to a 256-byte buffer into which the shell places the value of the variable. The value consists of a length byte and a string of ASCII characters. The value consists of a null string (that is, the length byte is 0x00) for an undefined variable.

The `index` parameter is an index number that you provide. Start with 0x01 and increment the number by 1 with each successive `READ_INDEXED` call until there are no more values in the variable table.

Possible errors

Errors for Memory Manager calls are described in the *Apple IIGS Toolbox Reference*.

INIT_WILDCARD (0x0109)

```
void INIT_WILDCARD(file, flags)
       char *file;
       int flags
```

This function provides to the APW Shell a filename that can include a wildcard character. The shell can then search for filenames matching the filename you specified when it receives a `NEXT_WILDCARD` command. This function accepts any filename, whether it includes a wildcard or not, and expands device names (such as `.D1/`), prefix numbers, and the double-period (..) before the filename is passed on to ProDOS 16. Therefore, you should call this function every time you want to search for a filename. Doing so will ensure that your routine supports all of the conventions for partial pathnames that the user expects from APW.

The `file` parameter is the address of a buffer containing a pathname or partial pathname that can include a wildcard character. Examples of such pathnames are as follows:

```
A=
/APW/MYPROGS/?.ROOT
.D2/HELLO
```

Important

The `file` parameter must be stored as a Pascal-style string: a length byte followed by the characters of the string.

When you execute a NEXT_WILDCARD call, the shell finds the next filename that matches the filename pointed to by file. If the wildcard character you specified was a question mark (?), then the filename is written to standard output and you are prompted for confirmation before the file is acted on or the next filename is found. The use of wildcard characters is described in "Using Wildcard Characters" in Chapter 2 of the *APW Reference.*

The flags parameter contains the prompting flags. If the most significant bit is set, prompting is not allowed; that is a question mark (?) is treated as if it were an equal sign (=). If the next-most significant bit is set and prompting is being used, only the first choice accepted by the user (that is, the first choice for which the user types Y in response to the prompt) is acted on. The second flag is for use with commands that can act on only one file, such as RENAME or EDIT.

Possible errors

Errors for ProDOS 16 and Memory Manager calls are described in the *Apple IIGS ProDOS 16 Reference* and the *Apple IIGS Toolbox Reference.* Use the ERROR function to get the error message.

NEXT_WILDCARD (0x010A)

```
char *NEXT_WILDCARD(nextfile)
       char *nextfile;
```

Once a filename that includes a wildcard has been suppled to the shell with an INIT_WILDCARD call, the NEXT_WILDCARD call causes the shell to find the next filename that matches the wildcard filename. For example, if the wildcard filename specified in INIT_WILDCARD were /APW/UTILITIES/XREF.?, then the first filename returned by the shell in response to a NEXT_WILDCARD call might be /APW/UTILITIES/XREF.ASM65816.

The nextfile parameter returns the address of the buffer to which the shell will return the next filename that matches a wildcard filename. The wildcard filename is the last one specified with an INIT_WILDCARD call. If there are no more matching filenames, or if INIT_WILDCARD has not been called, then the shell returns a null string (that is, a string with length zero). (See also the description of INIT_WILDCARD.)

Possible Errors None

GET_VAR (0x010B)

```
void GET_VAR(varname, value)
       char *varname, *value;
```

This function reads the string associated with a variable (that is, the value of the variable). The value returned is the one valid for the currently executing Exec file, or for the interactive command interpreter. Variables and Exec files are described in "Exec Files" in Chapter 3 of the *APW Reference.* Use the SET_VAR call to set the value of a variable.

The varname parameter is a pointer to a buffer that contains the name of the variable whose value you wish to read. The variable name consists of a length byte and a string of up to 255 ASCII characters.

The value parameter is a pointer to a 256-byte buffer into which the shell places the value of the variable. The value consists of a length byte and a string of ASCII characters. The value consists of a null string (that is, the length byte is 0x00) for an undefined variable.

Possible errors

None

EXECUTE (0x010D)

```
void EXECUTE(flag, comm)
       int flag;
       char *comm;
```

This function sends a command or list of commands to the APW Shell.

The flag parameter is used to execute an Exec file with an EXECUTE command; if no new variable table is defined, then variables defined by the list of commands modify the current variable table. If you set the most significant bit of this flag to 1 (binary), then a new variable table is not defined when the commands are executed. If this flag is set to 0x0000, a new variable table is defined for the list of commands being executed; the current variable table is not modified. Exec files, variables, and the EXECUTE command are described in "Exec Files" in Chapter 3 of the *APW Reference*.

The comm parameter is the address of the buffer in which you place the commands. If you include more than one command, separate the commands with semicolons (;) or carriage return characters (0x0D), the last command should end with a carriage return. The command string is a C string: it has no length byte and is terminated with a null character (0x00). Any output is sent to standard output.

If the shell variable {Exit} is not null and any command returns a nonzero error code, then any remaining commands are ignored. Error codes and shell variables are described in "Exec Files" in Chapter 3 of the *APW Reference*.

Possible errors

Any error returned from the last command or program executed by the list of commands executed.

DIRECTION (0x010F)

```
void DIRECTION(device, direct)
       int device,    *direct;
```

A program can use this function to find out whether command-line I/O redirection has occurred. This function can be used by a program to determine whether to send form feeds to standard output, for example.

The `device` parameter indicates which type of input or output you are inquiring about, as follows:

0x0000 Standard input
0x0001 Standard output
0x0002 Error output

The `direct` parameter indicates the type of redirection that has occurred, as follows:

0x0000 Console
0x0001 Printer
0x0002 Disk file

Possible errors

0x53 Parameter out of range

REDIRECT (0x0110)

```
void REDIRECT(device, app, file)
       int device, app;
       char *file;
```

This function instructs the shell to redirect input or output to the printer, console, or a disk file.

The `device` parameter indicates which type of input or output you wish to redirect, as follows:

0x0000 Standard input
0x0001 Standard output
0x0002 Error output

The app flag indicates whether redirected output should be appended to an existing file with the same filename, or the existing file should be deleted first. If append is 0, the file is deleted, if it is any other value, the output is appended to the file.

The `file` parameter is the address of a 65-byte-long buffer containing the filename of the file to or from which output is to be redirected. The filename can be any valid ProDOS 16 filename, a partial or full pathname, or the device names `.PRINTER` or `.CONSOLE`. The filename must be a Pascal string: that is, a length byte followed by the characters of the string.

Possible errors

0x53 Parameter out of range

Errors for ProDOS 16 calls are described in the *Apple IIGS ProDOS 16 Reference* and the *Apple IIGS Toolbox Reference.*

STOP (0x0113)

```
int STOP();
```

This function lets your application detect a request for an early termination of the program. The STOP flag is set when the keyboard buffer is read after the user presses Apple-period.

The STOP flag is set (0x0001) by the shell when it finds an Apple-period in the keyboard buffer. When a APW utility reads from the keyboard as standard input, the shell reads the keyboard buffer and passes the keys on to the utility. When standard input is not from the keyboard, the shell still checks the keyboard buffer for Apple-period whenever a STOP call is executed. The flag is cleared (0x0000) when the STOP call is executed, when the utility program is terminated, or if no Apple-period is found.

Possible errors

None

WRITE_CONSOLE (0x011A)

```
void WRITE_CONSOLE(ochar)
       int ochar;
```

This function writes a character to the Pascal console driver. The resulting output is not redirectable, so you can use this function to echo keyboard input and to send messages that must appear on the screen.

The `ochar` parameter is a 2-byte value specifying a character to write on the screen. The low byte of the value is sent to the Pascal console driver.

Possible errors

None

Appendix A

Calling Conventions

APW C uses two different function-calling conventions: C calling conventions and Pascal-compatible calling conventions.

C calling conventions

This section describes the normal C calling conventions. It explains how function parameters are passed, how function results are returned, and how registers are saved across function calls. This information is useful when writing calls between C and assembly language.

Parameters

Parameters to C functions are evaluated from right to left and are pushed onto the stack in the order they are evaluated: that is, they are pushed in reverse order. Characters, integers, and enumeration types are passed as 16-bit values. Long integers (`long`) are passed as 32-bit values; pointers and arrays are passed as 32-bit addresses. Types `float`, `double`, `comp`, and `extended` are passed as extended 80-bit values. Structures are also passed by value on the stack: their size is rounded up to a multiple of 16 bits (2 bytes). If rounding occurs, the unused storage has the highest memory address. The caller removes the parameters from the stack.

Function results

On the Apple IIGS, a function result is returned in registers: the low 16 bits are in the A register, and the high 16 bits are in the X register. A SANE-type result (that is, `float`, `double`, `comp`, and `extended`) or a structure result is returned as a pointer to an initialized static location; the pointer is in the A and X registers.

Register conventions

Only the stack pointer and the data bank register are preserved across function calls; no other registers are preserved. Tool calls have their own conventions for returning error codes in the A register. (These conventions are explained in "The Inline Declaration" in Chapter 4.)

Pascal-style calling conventions

This section describes the conventions used for calling functions that use Pascal-style calling conventions: these functions are declared with the keyword `pascal` and may have been written in any language. These conventions differ from the usual C calling conventions defined in Chapter 4.

Parameters

Parameters to Pascal-compatible functions are evaluated left to right: that is, in the order of the formal parameter list. The function first pushes space for the result (as shown in Table 3-2), and then pushes the parameters onto the stack in the order in which they are evaluated. Characters, integers, and enumeration types are passed as 16-bit values. Long integers (`long`) are passed as 32-bit values; pointers and arrays are passed as 32-bit addresses. SANE types and structures are passed on the stack. The size of a structure is rounded up to a multiple of 16 bits (2 bytes). If rounding occurs, the unused storage has the highest memory address. The function being called removes the parameters from the stack.

Function results

On the Apple IIGS, as on the Macintosh, a result of a Pascal-compatible function is returned on the stack. A SANE type or structure result is returned as a pointer to an initialized static location; the pointer is in the A and X registers.

Register conventions

Only the stack pointer and the data bank register are preserved across function calls; no other registers are preserved. Tool calls have their own conventions for returning error codes in the A register.

Appendix B

Files Supplied with APW C

APW C is intended for use with the Apple Programmer's Workshop. The files listed here are on the APW C release disk, which contains the C compiler, the Standard C Library, and the Apple IIGS Interface Library. These files may be used directly from the release disk or copied to a hard disk.

The files are listed indented under their respective directories, with comments.

APWC	APW C files
LANGUAGES	Compilers and assemblers
CC	APW C compiler
LIBRARIES	C libraries
CLIB	Standard C Library
START.ROOT	Initialization code
CINCLUDE	Standard C Library and Apple IIGS Toolbox include files
ADB.H	Apple Desktop Bus Manager
CONTROL.H	Control Manager
CTYPE.H	Character classification routines
DESK.H	Desk Accessory Manager
DIALOG.H	Dialog Manager
ERRNO.H	Error numbers
EVENT.H	Event Manager
FCNTL.H	File control
FONT.H	Font Manager
INTMATH.H	Fixed-Point Math
IOCTL.H	Device control
LINEEDIT.H	Line Editor
LIST.H	List Manager
LOADER.H	System Loader
LOCATOR.H	Tool locator
MALLOC.H	Memory allocation
MATH.H	Math functions
MEMORY.H	Memory Manager
MENU.H	Menu Manager
MISCTOOL.H	Miscellaneous Tools
NOTESEQ.H	Note Sequencer
NOTESYN.H	Note Systhesizer

PRINT.H	Print Manager
PRODOS.H	ProDOS interface
QDAUX.H	QuickDraw auxiliary
QUICKDRAW.H	QuickDraw II
SANE.H	SANE interface
SCHEDULER.H	Scheduler
SCRAP.H	Scrap Handler
SETJMP.H	Nonlocal transfer of control
SHELL.H	APW shell interface
SOUND.H	Sound Driver
STDFILE.H	Standard File Dialog Package
STDIO.H	Standard I/O Package
STDLIB.H	Miscellaneous Standard C Library declarations
STRING.H	String-conversion routines
STRINGS.H	String functions
TEXTTOOL.H	Text Tools
TYPES.H	Common defines and types
VALUES.H	SANE constants
VARARGS.H	Macros to handle variable number of arguments
WINDOW.H	Window Manager
SYSTEM	C system files
LOGIN	Log-in file
SYSCMND	Command file
INSTALL2	3.5-inch-disk install script
INSTALLHD	Hard-disk install file
SAMPLES	Sample programs
BONES	Longer application
MAKE	Build EXEC file
BONES.CC	Implements most of BONES
INIT.CC	Initializes tools
DATA.ASM	Data structures for windows and menus
STACKMIN.ASM	Allocates stack for BONES
LINK.BONES	Advanced linker instructions
DA	Desk accessory
IDLEHEADER.ASM	NDA identification section with pointers to four routines
CIDLE.C	Implements init, open, and close routines
USERIDLE.C	Implements action routine: customize to create your own
DB.ASM	Implements SaveDB() and RestoreDB()
MAKE	Build EXEC file
LINK.NDA	Advanced linker instructions
UPSTR	Short application
SAMPLEC	Implements the main event loop
SAMPLEA	Implements the uppercase function

Appendix C

Comparison with Macintosh Programmer's Workshop C

Apple IIGS Programmer's Workshop C is as closely related to Macintosh Programmer's Workshop C as differences between the two machines allow. The differences between the two languages are explained here.

Data types

The following data types are implemented differently in APW and MPW C:

Data Type	Size in bits	
	APW	MPW
`int`	16	32
`unsigned int`	16	32
`enum`	16	8, 16, or 32

The fact that

```
sizeof(int) != sizeof(char*)
```

creates many snares for the unwary. As a courtesy, `NULL` is defined in `stdio.h` to have the value `0L`.

Register variables

Register variables are not allocated in APW C due to the small number of registers available on the 65816.

Structured variables

Structures may be assigned, passed as parameters, and returned as function results in both versions of C. Byte-sized elements in structures are not padded to word or long-word boundaries. APW C allows equality comparison for structures; MPW C does not.

Pascal-compatible function declarations

A function or procedure written in Pascal (or written in assembly language following Pascal calling conventions) can be called from either MPW C or APW C. For example, the `DrawText` procedure is defined in Pascal as

```
PROCEDURE DrawText (textBuf: Ptr;
       firstByte, byteCount: INTEGER);
```

The MPW C syntax for such a declaration is

```
pascal void DrawText(textBuf, firstByte, byteCount)
       Ptr textBuf;
       short firstByte, byteCount;
       extern;
```

The APW C syntax for this declaration is

```
extern pascal void DrawText();
```

To make the APW C form more readable, you can list the parameters in a comment:

```
extern pascal void DrawText();
       /* Ptr textBuf;
       short firstByte, byteCount;
       extern; */
```

In addition, in MPW C, the word `extern` may be followed by a constant, which is interpreted as a 16-bit 68000 instruction that replaces the usual subroutine call (`JSR`) instruction in the calling sequence. This process allows direct traps to the Macintosh ROM, as shown here:

```
pascal void OpenPort(port)
       GrafPtr port;
       extern 0xA86F;
```

On the Apple IIGS, an `inline` declaration is used for declaring tool routines. Its syntax is

[`extern`] pascal [*result-type*] *func-name* `()  inline(`*m*`,` *n*`);`

This declaration says that the tool routine with tool-call number *n* and Tool-Locator entry point *m* can be called by the function name *func-name,* and returns a result of type *result-type*.

Preprocessor statements

A # alone as the first and only character of a line does not constitute a preprocessor directive that APW C understands. MPW blissfully ignores these; APW C complains.

APW C does not recognize preprocessor directives of the form

`#if defined(`*symbol*`)`

Dangling case in switch statements

If you have a dangling `case` in a `switch` statement, as in

```
switch(i) {case 1:  /* N.B. no statement here, just closing brace */ }
```

the APW C Compilerwill complain about an "error in expression", because K and R says that some kind of statement must follow the

`case` *constant-expression*:

In-line assembly-code declarations

An APW C program can contain in-line assembly code. Anywhere that a statement is legal, you can insert a series of assembly-language statements with this format:

`asm{` *assembly-statements*`}`

Anywhere that a function definition is legal, you can have a definition with this format:

`asm(` *external-name*`) {` *assembly-statements*`}`

This function can be called in the same way as a C function called *external-name*. Here *external-name* is the entry point of the segment containing the assembly-language code.

Appendix D

Library Index

The Library Index contains an index entry for each of the defines, types, enumeration literals, global variables, and functions defined in the Standard C Library.

- Column 1 contains an alphabetical list of the index entries.
- Column 2 specifies the type of declaration (for example, "function") for the index entry.
- Column 3 contains the library header under which documentation for the index entry can be found. If column 3 contains *(C)* following the library header—for example, *abs(C)*—look in Chapter 5, "The Standard C Library," which is organized alphabetically by library header. If column 3 contains *Shell,* look in Chapter 6, Shell Calls.

Identifier	Type	Manual section
abs	function	abs(C)
acos	function	trig(C)
asin	function	trig(C)
atan	function	trig(C)
atan2	function	trig(C)
atof	function	atof(C)
atoi	function	atoi(C)
atol	function	atoi(C)
BUFSIZ	define	setbuf(C)
c2pstr	function	string(C)
calloc	function	malloc(C)
ceil	function	floor(C)
cfree	function	malloc(C)
clearerr	function	ferror(C)
close	function	close(C)
cos	function	trig(C)
cosh	function	sinh(C)
creat	function	creat(C)
DIRECTION	function	Shell
dup	function	dup(C)
EACCES	define	Error(C)
EBADF	define	Error(C)
EBUSY	define	Error(C)
ecvt	function	ecvt(C)
EEXIST	define	Error(C)
EFAULT	define	Error(C)
EINVAL	define	Error(C)
EIO	define	Error(C)
ENFILE	define	Error(C)
ENODEV	define	Error(C)
ENOENT	define	Error(C)
ENOMEM	define	Error(C)
ENOSPC	define	Error(C)
ENOTDIR	define	Error(C)
ENXIO	define	Error(C)
EOF	define	stdio(C)
EROFS	define	Error(C)
ERROR	function	Shell
ESPIPE	define	Error(C)
ETXTBUSY	define	Error(C)
EXECUTE	function	Shell
exit	function	exit(C)
exp	function	exp(C)
fabs	function	floor(C)
faccess	function	faccess(C)
fclose	function	fclose(C)
fcntl	function	fcntl(C)
fcvt	function	ecvt(C)
fdopen	function	fopen(C)
feof	function	ferror(C)
ferror	function	ferror(C)
fflush	function	fclose(C)
fgetc	function	getc(C)
fgets	function	gets(C)
FILE	define	stdio(C)
fileno	function	ferror(C)
FIOBUFSIZE	define	ioctl(C)
FIODUPFD	define	ioctl(C)
FIOGETEOF	define	ioctl(C)
FIOGETMARK	define	ioctl(C)
FIOINTERACTIVE	define	ioctl(C)
FIOLSEEK	define	ioctl(C)
FIOREFNUM	define	ioctl(C)
FIOSETEOF	define	ioctl(C)
FIOSETMARK	define	ioctl(C)
floor	function	floor(C)
fmod	function	floor(C)
fopen	function	fopen(C)
fprintf	function	printf(C)
fputc	function	putc(C)
fputs	function	puts(C)
fread	function	fread(C)
free	function	malloc(C)
freopen	function	fopen(C)
frexp	function	frexp(C)
fscanf	function	scanf(C)
fseek	function	fseek(C)
ftell	function	fseek(C)
fwrite	function	fread(C)
F_AUX	define	faccess(C)
F_DELETE	define	faccess(C)
F_DUPFD	define	fcntl(C)
F_OPEN	define	faccess(C)
F_RENAME	define	faccess(C)
F_TYPE	define	faccess(C)
getc	function	getc(C)
getchar	function	getc(C)
getenv	function	getenv(C)
GetLInfoPB	type	Shell
gets	function	gets(C)
getw	function	getc(C)
GET_LANG	function	Shell
GET_LINFO	function	Shell
GET_VAR	function	Shell
hypot	function	hypot(C)
ioctl	function	ioctl(C)
isalnum	function	ctype(C)
isalpha	function	ctype(C)
isascii	function	ctype(C)
iscntrl	function	ctype(C)
isdigit	function	ctype(C)
isgraph	function	ctype(C)
islower	function	ctype(C)
isprint	function	ctype(C)
ispunct	function	ctype(C)
isspace	function	ctype(C)
isupper	function	ctype(C)
isxdigit	function	ctype(C)
ldexp	function	frexp(C)
lmalloc	function	malloc(C)
log	function	exp(C)
log10	function	exp(C)
longjmp	function	setjmp(C)
lseek	function	lseek(C)
malloc	function	malloc(C)
memccpy	function	memory(C)
memchr	function	memory(C)
memcmp	function	memory(C)
memcpy	function	memory(C)
memset	function	memory(C)
modf	function	frexp(C)
NULL	define	stdio(C)
onexit	function	onexit(C)
open	function	open(C)
O_APPEND	define	open(C)
O_BINARY	define	open(C)
O_CREAT	define	open(C)
O_EXCL	define	open(C)

Identifier	Type	Manual section
O_RDONLY	define	open(C)
O_RDWR	define	open(C)
O_TRUNC	define	open(C)
O_WRONLY	define	open(C)
p2cstr	function	string(C)
pow	function	exp(C)
printf	function	printf(C)
putc	function	putc(C)
putchar	function	putc(C)
puts	function	puts(C)
putw	function	putc(C)
qsort	function	qsort(C)
rand	function	rand(C)
read	function	read(C)
READ_INDEXED	function	Shell
realloc	function	malloc(C)
REDIRECT	function	Shell
rewind	function	fseek(C)
scanf	function	scanf(C)
setbuf	function	setbuf(C)
setjmp	function	setjmp(C)
setvbuf	function	setbuf(C)
SET_LANG	function	Shell
SET_LINFO	function	Shell
SET_VAR	function	Shell
SIGALLSIGS	define	signal(C)
sin	function	trig(C)
sinh	function	sinh(C)
sprintf	function	printf(C)
sqrt	function	exp(C)
srand	function	rand(C)
sscanf	function	scanf(C)
strcat	function	string(C)
strchr	function	string(C)
strcmp	function	string(C)
strcpy	function	string(C)
strcspn	function	string(C)
strlen	function	string(C)
strncat	function	string(C)
strncmp	function	string(C)
strncpy	function	string(C)
strpbrk	function	string(C)
strrchr	function	string(C)
strspn	function	string(C)
strtok	function	string(C)
strtol	function	strtol(C)
tan	function	trig(C)
tanh	function	sinh(C)
toascii	function	conv(C)
tolower	function	conv(C)
toupper	function	conv(C)
ungetc	function	ungetc(C)
unlink	function	unlink(C)
VERSION	function	Shell
write	function	write(C)
_exit	function	exit(C)
_IOFBF	define	setbuf(C)
_IOLBF	define	setbuf(C)
_IONBF	define	setbuf(C)
_IOSYNC	define	stdio(C)
_tolower	function	conv(C)
_toupper	function	conv(C)

Appendix E

ASCII Table

The ASCII table contains the equivalent ASCII values in decimal, octal, and hexadecimal for all characters in the Apple extended character set. The table is divided into columns of 32 characters each.

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
nul	0	0	0	sp	32	40	20	@	64	100	40	`	96	140	60
soh	1	1	1	!	33	41	21	A	65	101	41	a	97	141	61
stx	2	2	2	"	34	42	22	B	66	102	42	b	98	142	62
etx	3	3	3	#	35	43	23	C	67	103	43	c	99	143	63
eot	4	4	4	$	36	44	24	D	68	104	44	d	100	144	64
enq	5	5	5	%	37	45	25	E	69	105	45	e	101	145	65
ack	6	6	6	&	38	46	26	F	70	106	46	f	102	146	66
bel	7	7	7	'	39	47	27	G	71	107	47	g	103	147	67
bs	8	10	8	(	40	50	28	H	72	110	48	h	104	150	68
ht	9	11	9	)	41	51	29	I	73	111	49	i	105	151	69
lf	10	12	A	*	42	52	2A	J	74	112	4A	j	106	152	6A
vt	11	13	B	+	43	53	2B	K	75	113	4B	k	107	153	6B
ff	12	14	C	,	44	54	2C	L	76	114	4C	l	108	154	6C
cr	13	15	D	-	45	55	2D	M	77	115	4D	m	109	155	6D
so	14	16	E	.	46	56	2E	N	78	116	4E	n	110	156	6E
si	15	17	F	/	47	57	2F	O	79	117	4F	o	111	157	6F
dle	16	20	10	0	48	60	30	P	80	120	50	p	112	160	70
dc1	17	21	11	1	49	61	31	Q	81	121	51	q	113	161	71
dc2	18	22	12	2	50	62	32	R	82	122	52	r	114	162	72
dc3	19	23	13	3	51	63	33	S	83	123	53	s	115	163	73
dc4	20	24	14	4	52	64	34	T	84	124	54	t	116	164	74
nak	21	25	15	5	53	65	35	U	85	125	55	u	117	165	75
syn	22	26	16	6	54	66	36	V	86	126	56	v	118	166	76
etb	23	27	17	7	55	67	37	W	87	127	57	w	119	167	77
can	24	30	18	8	56	70	38	X	88	130	58	x	120	170	78
em	25	31	19	9	57	71	39	Y	89	131	59	y	121	171	79
sub	26	32	1A	:	58	72	3A	Z	90	132	5A	z	122	172	7A
esc	27	33	1B	;	59	73	3B	[	91	133	5B	{	123	173	7B
fs	28	34	1C	<	60	74	3C	\	92	134	5C	|	124	174	7C
gs	29	35	1D	=	61	75	3D	]	93	135	5D	}	125	175	7D
rs	30	36	1E	>	62	76	3E	^	94	136	5E	~	126	176	7E
us	31	37	1F	?	63	77	3F	_	95	137	5F	del	127	177	7F
Char	**Dec**	**Oct**	**Hex**	**Char**	**Dec**	**Oct**	**Hex**	**Char**	**Dec**	**Oct**	**Hex**	**Char**	**Dec**	**Oct**	**Hex**

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
Ä	128	200	80	†	160	240	A0	¿	192	300	C0	‡	224	340	E0
Å	129	201	81	°	161	241	A1	¡	193	301	C1	·	225	341	E1
Ç	130	202	82	¢	162	242	A2	¬	194	302	C2	‚	226	342	E2
É	131	203	83	£	163	243	A3	√	195	303	C3	„	227	343	E3
Ñ	132	204	84	§	164	244	A4	ƒ	196	304	C4	‰	228	344	E4
Ö	133	205	85	•	165	245	A5	≈	197	305	C5	Â	229	345	E5
Ü	134	206	86	¶	166	246	A6	Δ	198	306	C6	Ê	230	346	E6
á	135	207	87	ß	167	247	A7	«	199	307	C7	Á	231	347	E7
à	136	210	88	®	168	250	A8	»	200	310	C8	Ë	232	350	E8
â	137	211	89	©	169	251	A9	…	201	311	C9	È	233	351	E9
ä	138	212	8A	™	170	252	AA		202	312	CA	Í	234	352	EA
ã	139	213	8B	´	171	253	AB	À	203	313	CB	Î	235	353	EB
å	140	214	8C	¨	172	254	AC	Ã	204	314	CC	Ï	236	354	EC
ç	141	215	8D	≠	173	255	AD	Õ	205	315	CD	Ì	237	355	ED
é	142	216	8E	Æ	174	256	AE	Œ	206	316	CE	Ó	238	356	EE
è	143	217	8F	Ø	175	257	AF	œ	207	317	CF	Ô	239	357	EF
ê	144	220	90	∞	176	260	B0	–	208	320	D0		240	360	F0
ë	145	221	91	±	177	261	B1	—	209	321	D1	Ò	241	361	F1
í	146	222	92	≤	178	262	B2	“	210	322	D2	Ú	242	362	F2
ì	147	223	93	≥	179	263	B3	”	211	323	D3	Û	243	363	F3
î	148	224	94	¥	180	264	B4	‘	212	324	D4	Ù	244	364	F4
ï	149	225	95	µ	181	265	B5	’	213	325	D5	ı	245	365	F5
ñ	150	226	96	∂	182	266	B6	÷	214	326	D6	ˆ	246	366	F6
ó	151	227	97	∑	183	267	B7	◊	215	327	D7	˜	247	367	F7
ò	152	230	98	∏	184	270	B8	ÿ	216	330	D8	¯	248	370	F8
ô	153	231	99	π	185	271	B9	Ÿ	217	331	D9	˘	249	371	F9
ö	154	232	9A	∫	186	272	BA	⁄	218	332	DA	˙	250	372	FA
õ	155	233	9B	ª	187	273	BB	¤	219	333	DB	˚	251	373	FB
ú	156	234	9C	º	188	274	BC	‹	220	334	DC	¸	252	374	FC
ù	157	235	9D	Ω	189	275	BD	›	221	335	DD	˝	253	375	FD
û	158	236	9E	æ	190	276	BE	fi	222	336	DE	˛	254	376	FE
ü	159	237	9F	ø	191	277	BF	fl	223	337	DF	ˇ	255	377	FF
Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex

Appendix F

APW C Compiler Error Messages

The APW C Compiler can produce the following error messages:

Out of room (too many vars).
Illegal character.
Illegal preprocessor command.
Error in include command.
Include level cannot be > 6.
Error opening include file.
Error in define.
Too many defines.
Too few params in macro call.
Too many params in macro call.
Error in macro call.
Error in numerical constant.
Error in constant expression.
Error in struct or union def.
Error in declaration.
Error in parameter list.
Expected ';' missing.
Expected '}' missing.
Expected ')' missing.
Expected ']' missing.
Error in function definition.
Expected '(' missing.
Illegal statement.
Expected 'while' missing.
Expected ':' missing.
Error in goto statement.
Error in expression.
Not a legal storage class.
Redefining a union tag as a struct.
Redefining a struct tag as a union.
Argument must be integer.
Illegal initialization.
Expected '{' missing.
Cannot initialize union.
Undefined Identifier: .
Array or pointer type expected before '['.
Left side of assignment not an lvalue.

Array index must be integer type.
Argument for '*' must be pointer.
Arg before '.' must be union or struct lvalue.
Arg before '->' must be pointer to struct or union.
Function type expected before '('.
Bit field longer than unsigned size.
Bit field for type other than unsigned.
Constant expression must be integer.
Case not in switch statement.
Default not in switch statement.
More than one default for switch.
Illegal break.
Illegal continue.
Adding pointer to non-integer.
Subtracting pointer from non-pointer.
Subtracting pointers to things of different type.
Subtracting weird thing from a pointer.
Operand not a left value for *operator*.
Illegal union or struct usage.
Wrong number of initializers.
Address of register variable.
Can't define function here.
Syntax error in assembly code.
Invalid opcode.
Invalid addressing mode.
Expected comma missing.
Label not defined: .
'else' without matching 'if'.
expected string missing.
Illegal operation size.
Undefined or improperly used field.
Structure or union can't contain self.
Auto vars or constants only with this address mode.
Error in line command.
No defines allowed in in-line assembly.
Error in #undef.
Pointers do not point to same type object.
Bad token.
Unexpected semicolon.
An ostrap must be of type function.
Multiply defined label: .
Too many local variables.
Can't cast a non-lval into an array.
Declared argument .
String too long.
Syntax error in segment command.
Missing endif.
Missing close of comment.
Define recursively defined or too complex.
Error writing output file (Disk full?).
& before function or array name: ignored.
Bitfields not allowed in union.
Can't take the address of a bitfield.
Duplicate case in switch.
Can't pass a function as a parameter.
Zero or negative subscript.
newline in string or char constant.
void type not allowed in expression.

Glossary

*** (asterisk):** A 32-bit pointer data type.

absolute code: Program code that must be loaded at a specific address in memory and never moved.

absolute segment: A segment that can be loaded only at one specific location in memory. Compare with **relocatable segment.**

accumulator: The register in the 65C816 microprocessor of the Apple IIGS used for most computations.

address: A number that specifies the location of a single **byte** of memory. Addresses can be given as decimal or hexadecimal integers. The Apple IIGS has addresses ranging from 0 to 16,777,215 (in decimal) or from $00 00 00 to $FF FF FF (in hexadecimal). A complete address consists of a 4-bit **bank** number ($00 to $FF) followed by a 16-bit address within that bank ($00 00 to $FF FF).

advanced linker: The APW Linker running a file of LinkEd commands.

Apple key: A modifier key on the Apple IIGS keyboard, marked with an Apple icon. It performs the same functions as the *Open Apple* key on standard Apple II machines.

Apple II: A family of computers, including the original Apple II, the Apple II Plus, the Apple IIe, the Apple IIc, and the Apple IIGS.

`AppleIIgs`: A predefined constant identifying C code written for the AppleIIGS—in particular, for APW C.

Apple IIGS Interface Libraries: A set of **interfaces** that enable you to access toolbox routines from C.

Apple IIGS Toolbox: An extensive set of routines that facilitate writing desktop applications and provide easy program access to many Apple IIGS hardware and firmware features.

`APW`: A predefined constant identifying C code written for the APW C Compiler as opposed to another C compiler.

APW Linker: The linker supplied with APW.

APW Shell: The programming environment of the Apple IIGS Programmer's Workshop. It lets you edit programs, manipulate files, and execute programs.

application: A program (such as the APW Shell itself) that talks to ProDOS and the Toolbox directly, and can be exited via the `Quit` call.

assembler: A program that produces object files from source files written in assembly language.

automatic variable: A dynamic local variable that comes into existence when a function is called and that disappears when it is exited.

bank: A 64K (65,536-byte) portion of the Apple IIGS internal memory. An individual bank is specified by the value of one of the 65C816 microprocessor's bank registers.

buffer: An area of memory allocated for reading from or writing to a file.

catalog: See **directory**.

carriage return character (`\r`): A control code (ASCII 13) generated by the Return key; in APW C, equal to **newline (`\n`).**

`char`: An 8-bit character data type whose range is 0 to 255; the same as **`unsigned char`** in APW C.

character: Any symbol that has a widely understood meaning and thus can convey information. Some characters—such as letters, numbers, and punctuation—can be displayed on the monitor screen and printed on a printer. Most characters are represented in the computer as 1-byte values.

code segment: An object segment that consists mainly of code. Code segments are provided for programs that differientiate between code and data segments.

command: In the Standard C Library, a parameter that tells a function which of several actions to perform; in the APW Shell, a word that tells APW which utility to execute.

command interpreter: A program that interprets and executes commands; specifically, the APW shell.

comp: A 64-bit SANE data type with signed integral values and one NaN.

compiler: A program that produces object files from source files written in a high-level language such as C.

conditional compilation: Use of preprocessor commands (`#if`, `#ifdef`, `#ifndef`, `#else`, `#endif`) to vary the output depending on compile-time conditions.

C SANE Library: A set of routines that provide extended-precision mathematical functions.

current language: The APW language type that is assigned to a file opened by the APW Editor. If an existing file is opened, the current language changes to match that of the file.

current prefix: The prefix that is used by the APW Shell if a partial pathname is used.

data segment: An object segment that consists primarily of data. Data segments are provided for programs that differentiate between code and data segments.

debugger: A shell utility that lets you step through a program and examine memory as you go.

denormalized number: A nonzero number that is too small for normalized representation.

desk accessory: A program that is accessed from the Apple menu and shares its run-time environment with an application, a utility, or another desk accessory.

desktop user interface: The visual interface between the computer and the user—the menu bar and the gray (or solid-colored) area on the screen. In many applications the user can have a number of **documents** on the desktop at the same time.

diagnostic output: A file used to report errors and diagnostic information; generally merged with **standard output,** but can be redirected; in APW C, synonymous with **standard error**.

directory: A file that contains a list of the names and locations of other files stored on a disk. Directories are either **volume directories** or **subdirectories**. A directory is sometimes called a *catalog*.

direct page: A page (256 bytes) of bank $00 of Apple IIGS memory, any part of which can be addressed with a short (1 byte) address because its high address byte is always $00 and its middle address byte is the value of the 65C816 processor's direct register. Coresident programs or routines can have their own direct pages at different locations. The direct page corresponds to the 6502 processor's **zero page**. The term *direct page* is often used informally to refer to the lower portion of the **direct-page/stack space**.

direct-page/stack space: A portion of bank $00 of Apple IIGS memory reserved for a program's **direct page** and **stack**. Initially, the 65C816 processor's **direct register** contains the base address of the space, and its **stack register** contains the highest address. In use, the stack grows downward from the top of the direct-page/stack space, and the lower part of the space contains direct-page data.

direct register: A hardware register in the 65C816 processor that specifies the start of the direct page.

dispose: To deallocate a memory block permanently. The Memory Manager disposes of a memory block by removing its master pointer. Any handle to that pointer will then be invalid. Compare with **purge.**

double: A 64-bit floating-point data type with IEEE double precision.

dynamic segment: A segment that can be loaded and unloaded during execution as needed. Compare with **static segment.**

editor: A shell utility for editing source files.

enum: An enumerated data type of 8, 16, or 32 bits depending on the range of the enumerated literals.

environment: In SANE, consists of rounding direction, rounding precision, exception flags, and halt settings; in APW, consists of exported variables and other features of the Integrated Environment.

event: A notification to an application of some occurrence (such as an interrupt generated by a keypress or mouse click) to which the application may want to respond.

event-driven program: A kind of program that responds to user inputs in real time by repeatedly testing for **events.** An event-driven program does nothing until it detects an event.

exception: A condition in the SANE environment that can cause a program halt.

Exec file: A file containing APW commands that are executed as if typed on the keyboard.

`exit` function: A function that is registered with `onexit` for execution when the program terminates.

`extended`: An 80-bit floating-point data type with IEEE extended precision; used in C for all intermediate results.

external reference: A reference to a symbol that is defined in another segment. External references must be to global symbols.

fatal error: An error serious enough that the computer must halt execution.

field: A string of ASCII characters or a value that has a specific meaning to some program. Fields may be of fixed length, or may be separated from other fields by field delimiters. For example, each parameter in a segment header constitutes a field.

file-buffered: A buffer style in which characters sent to an output I/O function are queued and written as a block.

file descriptor: A file reference number returned by a `creat` or `open` call.

filename: The string of characters that identifies a particular file within a disk **directory**. ProDOS 16 filenames can be up to 15 characters long, and can specify directory files, subdirectory files, text files, source files, object files, load files, or any other ProODS 16 file type. Compare with **pathname**.

file pointer: A pointer to the next byte to be read or written in a **stream**.

file type: An attribute in a ProDOS 16 file's directory entry that characterizes the contents of the file and indicates how the file may be used. On disk, filetypes are stored as numbers; in a directory listing, they are often displayed as three-character mnemonic codes.

FILE variable: A variable containing information about a stream, including the file descriptor and buffer size, location, and style.

`float`: A 32-bit floating-point data type with IEEE single precision.

flush: Write out the contents of a buffer.

format character: A character that defines the interpretation of the input field in the `scanf` call.

full pathname: The complete name by which a file is specified. A full pathname always begins with a slash (/) because a volume directory name always begins with a slash. See **pathname**.

function: In C, any subroutine, whether or not it returns a value. Equivalent to the Pascal word *procedure;* the Pascal word *function* means a subroutine that returns a value.

global label: A symbolic identifier in an object segment, which the linker enters into the relocation dictionaly and the loader replaces with an absolute address.

global symbol: A label in a code segment that is either the name of the segment or an entry point to it. Global symbols may be referenced by other segments. Compare with **local symbol**.

handle: See **memory handle**.

header file: A file whose contents will be included in the source file at compile time—it contains function declarations, macros, types, and `#define` directives used by the compiler. (Also called an *include file*)

hexadecimal: The base–16 system of numbers, using the ten digits 0 through 9 and the six letters A through F. Hexadecimal numbers can be converted easily and directly to binary form, because each hexadecimal digit corresponds to a sequence of 4 bits. In C manuals hexadecimal numbers are usually preceded by a `0x`.

high–level language: A programming language that is relatively easy for people to understand. A single statement in a high–level language typically corresponds to several instructions of machine language. Compare **low–level language.**

image: A representation of the contents of memory. A code image consists of machine-language instructions or data that may be loaded unchanged into memory.

include file: A file whose contents will be included with the source file at compile time—it contains function declarations, macros, types, and `#define` directives used by the compiler.

infinity: A SANE representation of mathematical ∞.

`int`: A 16-bit integer data type whose range is –32,768 to 32,767.

interface: The compile-time and run-time linkage between your C program and toolbox routines.

Jump Table: A table contructed in memory by the System Loader from all Jump Table segments encountered during a load. The Jump Table contains all references to dynamic segments that may be called during execution of the program.

K: 1024 bytes

language.command: A command that changes the APW current language.

library dictionary segment: The first segment of a library file; it contains a list of all the symbols in the file together with their locations in the file. The linker uses the library dictionary segment to find the segments it needs.

library file: A file produced by MAKELIB program from object files, generally ones containing functions useful to a number of programs. It can be searched by the linker for necessary functions, but more quickly than an object file.

line-buffered: A **buffer** style in which each line of output is queued for writing as soon as a newline character is written.

LinkEd: A command language that can be used to control the APW Linker.

linker: A program that combines files generated by compilers and assemblers, resolves all symbolic references, and generates a file that can be loaded into memory and executed.

load file: A file that can be loaded into memory, one load segment at a time, by the System Loader.

load segment: A part of a load file corresponding (in C) to one or more functions. **Object segments** are assigned to load segments at compile time by means of the `overlay` command or at link-time by LinkEd commands.

local symbol: A label defined only within an individual segment. Other segments cannot access the label. Compare with **global symbol**.

`long`: A 32-bit integer data type whose range is –2,147,483,648 to 2,147,483,647.

loop: A section of a program that is executed repeatedly until a limit or condition is met, such as an index variable's reaching a specified ending value.

low-level language: A programming language, such as assembly language, that is relatively close to the form the computer's processor can execute directly. One statement in a low-level language corresponds to a single machine-language instruction. Compare **high-level language.**

`main`: The name of the function that is the entry point for every C program.

main segment: The first segment in the initial load file of a program. It is loaded first and never removed from memory until the program terminates.

MakeLib utility: A program that creates library files from object files.

Mark: The current position in an open file. It is the point in the file at which the next read or write operation will occur.

memory block: See **block.**

memory handle: The identifying number of a particular block of memory. A memory handle is a pointer to a master pointer to the memory block.

memory image: A portion of a disk file or segment that can be read directly into memory.

Memory Manager: A program in the Apple IIGS Toolbox that manages memory use. The Memory Manager keeps track of how much memory is available, and allocates memory **blocks** to hold program segments or data.

memory-resident: (adj) (1) Stored permanently in memory as firmware (ROM). (2) Held continually in memory even while not in use. For example, ProDOS is a memory-resident program.

Memory Segment Table: A linked list in memory, created by the loader, that allows the loader to keep track of the segments that have been loaded into memory.

movable: A memory block attribute, indicating that the Memory Manager is free to move the block; opposite of *fixed*. Only **position-independent** program segments may be in movable memory blocks. A block is made movable or fixed through Memory Manager calls.

NaN: Not a Number; a SANE representation produced when an operation cannot yield a meaningful result.

native mode: The 16-bit operating state of the 65C816 processor.

newline character (\n): A control code that advances print position or cursor to the left margin of next output line; in APW C, same as **carriage return (\r).**

normalized number: A floating-point number that can be represented with a leading significand bit of 1.

number class: In SANE, a floating-point number can be characterized as either zero, normalized, denormalized, infinity, or NaN.

numeric environment: In SANE, the rounding direction, rounding precision, halt enables, and exception flags.

object segment: A part of an object file corresponding (in C) to a single function.

object file: The output from an assembler or compiler and the input to the linker. In APW, an object file contains both machine-language instructions and instructions for the linker. Compare with **load file.**

object module format (OMF)**:** The general format used in object files, library files, and load files.

object segment: A segment in an object file.

OMF: Object module format.

OMF file: Any file in object module format.

page: (1) A portion of Apple IIGS memory that is 256 bytes long and that begins at an address that is an even multiple of 256. A memory block whose starting address is an even multiple of 256 is said to be *page aligned*. (2) An area of main memory containing text or graphical information being displayed on the screen.

parameter: A value passed to or from a command, function, or other routine.

Pascal-style function: A function using Pascal-style calling conventions that can be declared in C using the `pascal` specifier.

partial assembly: A procedure whereby only specific segments of a program are assembled. If you have performed one full assembly followed by one or more partial assemblies on a program, the linker extracts only the latest version of each object segment to be included in the load file.

partial compile: A procedure whereby only specific segments of a program are compiled. If you have performed one full assembly followed by one or more partial compiles on a program, the linker extracts only the latest version of each object segment to be included in the load file.

partial pathname: A **pathname** that includes the **filename** of the desired file but excludes the volume directory name (and possibly one or more of the subdirectories in the path). It is the part of a pathname following a **prefix**—a prefix and a partial pathname together constitute a **full pathname**. A partial pathname does not begin with a slash because it has no volume directory name.

patch: To replace one or more bytes in memory or in a file with other values. The address to which the program must jump to execute a subroutine is *patched* into memory at load-time when a file is **relocated**.

pathname: The full name of a file, including its volume name and directory names.

pointer: A memory address at which a particular item of information is located. For example, the 65C816 Stack register contains a pointer to the next available location on the stack.

position-independent: Code that is written specifically so that its execution is unaffected by its position in memory. It can be moved without needing to be relocated.

position-independent segment: A load segment that is movable when loaded in memory.

prefix: A portion of a **pathname** starting with a volume name and ending with a subdirectory name. It is the part of a full pathname that precedes a **partial pathname**—a prefix and a partial pathname together constitute a full pathname. A prefix always starts with a slash (/) because a volume directory name always starts with a slash.

preprocessor: Part of the C compiler that provides file inclusion, macro substitution, and conditional compilation.

preprocessor symbol: One of a set of constants defined to be 1, equivalent to writing "`#define` *symbol* `1`" at the beginning of the source file.

ProDOS: A family of disk operating systems developed for the Apple II family of computers. *ProDOS* stands for *Professional Disk Operating System*, and includes both ProDOS 8 and ProDOS 16.

ProDOS 8: A **disk operating system** developed for standard Apple II computers. It runs on 6502-series microprocessors. It also runs on the Apple IIGS when the 65C816 processor is in 6502 **emulation mode**.

ProDOS 16: A **disk operating system** developed for 65C816 **native mode** operation on the Apple IIGS. It is functionally similar to **ProDOS 8** but more powerful.

purge: To deallocate a memory block temporarily. The Memory Manager purges a block by setting its master pointer to 0. All handles to the pointer are still valid, so the block can be reconstructed quickly. Compare with **dispose.**

purgeable: A memory block attribute, indicating that the Memory Manager may purge the block if it needs additional memory space. Purgeable blocks have different **purge levels**, or priorities for purging; these levels are set by Memory Manager calls.

RAM Disk: A portion of memory (RAM) that appears to the operating system to be a disk volume. Files in a RAM disk can be accessed much faster than the same files on a floppy disk or hard disk.

`register` variable: An **automatic variable** that is allocated to a register; not used by APW C Compiler because the 65C816 has only a few registers.

relocate: To modify a file or segment at load time so that it will execute correctly at the location in memory at which it is loaded. Relocation consists of **patching** the proper values into address operands. The loader relocates load segments when it loads them into memory. See also **relocatable code.**

relocatable code: Program code that includes no absolute addresses, and so can be relocated at load time.

relocatable segment: A segment that can be loaded at any location in memory. A relocatable segment can be static, dynamic, or position independent. A load segment contains a **relocation dictionary** that is used to recalculate the values of location-dependent addresses and operands when the segment is loaded into memory. Compare with **absolute segment.**

relocation dictionary: A portion of a load segment that contains relocation information necessary to modify the memory image immediately preceding it. When the memory image part of the segment is loaded into memory, the relocation dictionary is processed by the loader to calculate the values of location-dependent addresses and operands. Relocation dictionaries also contain the information necessary to transfer control to external references.

reference: The name of a segment or entry point to a segment; same as *symbolic reference;* to refer to a symbolic reference or to use one in an expression or as an address.

resolve: To find the segment and offset in a segment at which a symbolic reference is defined. When the linker resolves a reference it creates an entry in a **relocation dictionary** that allows the loader to **relocate** the reference at load time.

root filename: The filename of an object file minus any filename extensions added by the assembler or compiler. For example, a program that consists of the object files `MYPROG.ROOT`, `MYPROG.A`, and `MYPROG.B` has the root filename `MYPROG`.

run-time library file: A load file containing program segments--each of which can be used in any number of programs--that the system loader loads dynamically when they are needed.

scanset: A set of characters allowed in a file scanned by the `scanf` call.

segment: A component of an OMF file, consisting of a header and a body. In object files, each segment incorporates one or more subroutines. In load files, each segment incorporates one or more object segments.

segment body: That part of a segment that follows the **segment header**, and that contains the program code, data, and relocation information for the segment.

segment header: The first part of a program segment, containing such information as the segment name and the length of the segment.

segment kind: See **segment type.**

segment number: A number corresponding to the relative position of the segment in a file, starting with 1.

segment type: A classification of a segment based on its purpose, contents, and internal structure, as defined in the object module format. The segment type is specified by the KIND field in the segment header.

shell: A program that provides an operating environment for other programs, and that is not removed from memory when the those programs are running. For example, the APW Shell provides a command processor interface between the user and the other components of APW, and remains in memory when APW utility programs are running.

shell call: A request from a program to the APW Shell to perform a specific function.

shell application: A type of program, such as a compiler or shell command, that runs under the APW Shell; called a **tool** in MPW.

shell load file: A load file designed to be run under a shell program; shell load files are ProDOS 16 file type $B5.

`short`: A 16-bit integer data type whose range is –32,768 to 32,767.

signal: A software interrupt that causes a program to be temporarily diverted from its normal execution sequence.

65C816: The microprocessor used in the Apple IIGS.

source file: An ASCII file consisting of instructions written in a particular language, such as C or assembly language. An assembler or compiler converts source files into **object files**.

stack: A list in which entries are added (pushed) and removed (pulled) at one end only (the top of the stack), causing them to be removed in last-in, first-out (LIFO) order. The term *the stack* usually refers to the top portion of the **direct-page/stack** space; the top of this stack is pointed to by the 65C816's *Stack register.*

Standard C: A de facto standard based on the most widely used implementation, the Berkleley VAX Portable C Compiler.

Standard C Library: A collection of routines for I/O, string manipulation, data conversion, memory management, and Integrated Environment support.

standard error: A file used to report errors and diagnostic information; generally merged with **standard output,** but can be redirected; in APW C, synonymous with **diagnostic output.**

standard input: The standard input stream; generally the keyboard, but can be redirected so that input is taken from a file or device.

standard linker: The APW Linker called directly by a shell command like `LINK`.

standard output: The standard output stream; generally the screen but can be redirected so that input is sent to a file or device.

static segment: A segment that is loaded at program boot-time, and is not unloaded or moved during execution. Compare with **dynamic segment.**

stream: A file with associated buffering.

string: An item of information consisting of a sequence of text characters (a *character string*), or a sequence of bits or bytes.

`struct`: A record data type.

subdirectory: A directory within a directory; a file (other than the volume directory) that contains the names and locations of other files. Every ProDOS 16 directory file is either a volume directory or a subdirectory.

symbol: A character or string of characters that represents an address or numeric value; a symbolic reference or a variable.

symbolic reference: A name or label that is used to refer to a location in a program, such as the name of a subroutine. When a program is linked, all symbolic references are **resolved**; when the program is loaded, actual memory addresses are **patched** into the program to replace the symbolic references.

symbol table: A table of symbolic references created by the linker when it links a program. The linker uses the symbol table to keep track of which symbols have been resolved. At the conclusion of a link, you can have the linker print out the symbol table.

System Loader: The program that relocates load segments and loads them into Apple IIGS memory. The System Loader works closely with ProDOS 16 and the Memory Manager.

system program: (1) A software component of a computer system that supports **application programs** by managing system resources such as memory and I/O devices. Also called *system software.* (2) Under ProDOS 8, a stand-alone and potentially self-booting application. A ProDOS 8 system program is of file type $FF; if it is self-booting, its filename has the extension `.SYSTEM`.

token: The smallest unit of information processed by a compiler or assembler. In C, for example, a function name and a left bracket ([) are tokens.

tool: An Apple IIGS Toolbox routine.

toolbox: A collection of built-in routines on the Apple IIGS that programs can call to perform many commonly needed functions. Functions within the toolbox are grouped into **tool sets**.

tool set: A related group of (usually firmware) routines, available to applications and system software, that perform necessary functions or provide programming convenience. The Memory Manager, the System Loader, and Quickdraw II are tool sets.

utility: In general, an application program that performs a relatively simple function or set of functions, such as copying or deleting files. An APW utility is a program that runs under the APW Shell, and that performs a function not handled by the shell itself. MAKELIB is an example of a APW utility.

unbuffered: A buffer style that does not use a buffer for I/O; reading and writing is done one character at a time.

unload: To remove a load segment from memory. To unload a segment, the System Loader does not actually "unload" anything; it calls the Memory Manager to either **purge** or **dispose** of the memory block in which the code segment resides. The loader then modifies the Memory Segment Table to reflect the fact that the segment is no longer in memory.

unordered: The result of a comparison with a NaN; even identical NaNs compare unordered.

`unsigned char`: An 8-bit character data type whose range is 0 to 255; the same as **`char`** in APW C.

`unsigned int`: A 16-bit integer data type whose range is 0 to 65,535.

`unsigned long`: A 32-bit integer data type whose range is 0 to 4,294,967,295.

`unsigned short`: A 16-bit integer data type whose range is 0 to 65,535.

`void`: A data type used to declare a function that does not return a value.

volume: An item that stores data; the source or destination of information. A volume has a name and a volume directory with the same name. Volumes typically reside in **devices**; a *device* such as a floppy-disk drive may contain one of any number of *volumes* (disks).

volume directory: The main directory file of a volume. It contains the names and locations of other files on the volume, any of which may themselves be directory files (called **subdirectories**). The name of the volume directory is the name of the volume. The pathname of every file on the volume starts with the volume directory name.

wildcard character: A character that may be used as shorthand to represent a sequence of characters in a pathname. In APW, the equal sign (=) and the question mark (?) can be used as wildcard characters.

word: A group of bits that is treated as a unit. For the Apple IIGS, a word is 16 bits (2 bytes) long.

WD65816: A predefined symbol identifying C code written to run on the Western Design Center 65SC816 as opposed to another microprocessor.

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS computer when running a standard Apple II program). Because the high-order byte of any address in this part of memory is zero, only a single byte is needed to specify a zero-page address. Compare **direct page.**

Index

A

abs 5-5
absolute code 1-7
acos 5-61
advanced linker 1-3
ALINK 6-6
app 6-11
append 6-11
#append 4-23
appending files 2-16
AppleIIGS 4-6
Apple IIGS, technical manuals ix–xiii
Apple IIGS Debugger 1-16
Apple IIGS Programmer's Workshop. See APW
Apple IIGS Toolbox xi, 1-2, 1-17, 4-12
APW 4-6
APW Assembler 1-15, 1-16, 2-12, 2-17, 3-5, 4-9–12
APW C 1-3–4
  assignment operators 4-22
  bit fields 4-21
  calling conventions A-1–2
  dynamic segments 1-12–13
  evaluation order 4-21
  files supplied B-1–2
  implementations 4-19–25
  installing 2-2–3
  libraries 1-17
  library files 1-13
  MPW C compared C-1–3
  numeric constants 4-3
  parameters 4-16–18
  Pascal-compatible function declarations C-2
  Pascal-style functions 4-12–16
  program interactions 1-14–16
  program segmentation 1-8–12
  register variables 4-5
  relocatable load files 1-6–7
  reserved symbols 4-6
  running on 3-5-inch disks 2-3–4
  SANE extensions 4-6–9
  string substitutions 4-21
  structures 4-5
  variable names 4-2–3
  writing desk accessories 3-8–10
APW C Compiler 1-16, 2-1–18, 4-19
  compilation process 2-5
  error messages 2-6, F-1–2
  limitations 4-22
  shell commands 2-6–17
  suspending/aborting compilation 2-6
  variable allocation 4-19
APW C disk, backing up 2-2
APW Editor 1-3, 1-15, 2-9, 2-10, 3-2, 3-5, 6-4, 6-6
APW Linker 1-3, 1-7, 1-16, 2-14, 3-3, 3-6, 6-5
APW Shell xii, 1-2, 1-15, 1-17, 5-15, 5-28, 5-46
  calls 6-1–12
  command interpreter 2-8
arg 5-17, 5-19, 5-31, 5-39, 5-40, 5-41
argc 4-23
argv 4-23
array indexing 4-19–20
~arrays 1-9–10, 1-12, 4-25
ASCII table E-1–3
asin 5-61
ASM65816 1-15–16
ASML 2-7, 6-4, 6-6
ASMLG 2-7
ASSEMBLE 2-10
assembly code, in-line 4-9–12
assignment operators 4-22
asterisk (*) 5-39, 5-47
atan 5-61
atan2 5-61
atof 5-6
atoi 5-7
atoi 5-60
atol 5-60
auto 4-10, 4-23

B

base 5-44, 5-60
blank 5-40
buf 5-46, 5-64
buffer initialization 5-56
BUFSIZ 5-51, 5-56

C

CC 1-15, 2-6, 2-9
C compiler xiii. See also APW C Compiler
ceil(x) 5-21
cfree 5-35
CHANGE 2-6, 2-9
circumflex (^) 5-49
classic desk accessories 3-8
clearerr 5-20, 5-55
CLIB 4-6, 4-8, 4-13, 6-2
close 5-8, 5-18
cmd 5-17, 5-19
CMPL 2-7, 2-9, 3-3
CMPLG 1-16, 2-7, 2-9, 2-15
code 2-17
code-generation memory model 4-24–25
code segments, relocatable 1-6–7
comm 6-10
command interpreter 1-2
  APW Shell 2-8
comp 4-7, 4-8, 4-9, 5-40, 5-41, A-1
COMPACT 3-2, 3-7
compar 5-44
COMPILE 2-5, 2-7, 2-10–13, 2-16, 3-2
Control Panel 3-8
conv 5-9
cos 5-61
cosh 5-54
creat 5-10, 5-55, 5-64
CRUNCH 1-16
c2pstr 5-59
ctype 5-11–12
current language 1-15, 3-3
current prefix 2-3

D

data 2-17
data segments 1-9
data types
  APW C 4-2–3
  global and extended 4-16
  MPW C C-1
  parameter and result 4-15–16
decpt 5-14
Desk Manager 3-8, 3-9
Desktop Bus xii
desktop user interface xii
dest 5-36
destStr 5-58, 5-59
device 6-11
dfile 6-4, 6-5
direct 6-11
DIRECTION 6-2, 6-11
double 4-7, 4-8, 4-9, 5-40, 5-41, A-1
double period (..) 5-38, 6-8
DrawText 4-13
dup 5-13, 5-55, 5-64
dynamic 1-13, 4-23
dynamic segments 1-12–13, 2-17

E

EACCES 5-3
EBADF 5-3
EBUSY 5-3
ecvt 5-14
EDIT 2-6, 2-13, 6-9
editor 1-2
EEXIST 5-4
EFAULT 5-3
EINVAL 5-4, 5-31, 5-32
EIO 5-3
elsize 5-35, 5-44
EMFILE 5-4
ENFILE 5-4
ENODEV 5-4
ENOENT 5-3
ENOMEM 5-3
ENOSPC 5-4
ENOTDIR 5-4
enum 4-4–5
enumeration types 4-4–5
env 5-53
environment 5-28
ENXIO 5-3
equal sign (=) 6-9
EROFS 5-4
errno 5-3, 5-4, 5-31, 5-32, 5-38, 5-46, 5-63, 5-64
ERROR 6-2, 6-7
error messages, APW C Compiler 2-6, F-1–2
ETXTBSY 5-4
event-driven program xi
EXECUTE 6-2, 6-3, 6-10
exit 5-15, 5-37, 5-57
exit () 4-23
exp 5-16
exp(x) 5-16
extended 4-7, 4-8, 4-9, 5-40, 5-41, A-1

F

fabs(x) 5-21
faccess 5-17
fclose 5-57
fcntl 5-19, 5-55, 5-64
fcvt 5-14
fdopen 5-18, 5-23, 5-55, 5-56
feof 5-20, 5-27, 5-55
ferror 5-20, 5-27, 5-55
fflush 5-18
fgetc 5-27, 5-55
fgets 5-29, 5-55
fildes 5-8, 5-13, 5-19, 5-31, 5-33, 5-46, 5-64
file 6-8, 6-9, 6-11
FILE 5-56, 5-57
__FILE__ 4-6
filename 5-22–23, 5-23, 5-38
fileno 5-20, 5-55
findChars 5-59
FIOBUFSIZE 5-31, 5-32, 5-56
FIODUPFD 5-32
FIOGETEOF 5-31
FIOGETMARK 5-31
FIOINTERACTIVE 5-31, 5-32, 5-56
FIOLSEEK 5-32
FIOREFNUM 5-31, 5-32
FIOSETEOF 5-31
FIOSETMARK 5-31
flag 6-10
flags 6-9
float 4-7, 4-8, 4-9, 5-40, 5-41, A-1
floor 5-21
fmod 5-21
fopen 5-22–23, 5-55, 5-56
format 5-39, 5-47
format characters 5-47–49
fprintf 5-39, 5-41, 5-55
fputc 5-42, 5-55
fputs 5-43, 5-55
fread 5-2, 5-24, 5-46, 5-55
free 5-34
freopen 5-18, 5-23, 5-55, 5-56
frexp 5-25
fscanf 5-47, 5-50, 5-55
fseek 5-23, 5-26
ftell 5-26
func 5-37
function 1-8
fwrite 5-2, 5-24, 5-55

G

getc 5-27, 5-55, 5-62
getchar 5-27, 5-55
getenv 5-28
GET_LANG 6-2, 6-6
GET_LINFO 6-2, 6-3–6
GetLInfoPB 6-3, 6-4
gets 5-29, 5-55
GET_VAR 6-2, 6-7, 6-9–10
getw 5-27, 5-55
global labels 2-5
~globals 1-9–10, 1-12, 4-25
global symbols 1-13

H

header file 1-6
hypot 5-30

I, J

include file 1-6
include-file search rules 2-18
index 5-59, 6-8
INIT_WILDCARD 6-2, 6-8–9
inline 4-13
I/O buffering 5-55–57
ioctl 5-31–32
isalnum 5-11
isalpha 5-11
isascii 5-11
iscntrl 5-11
isdigit 5-11
isgraph 5-11
islower 5-11
isprint 5-11
ispunct 5-11
isspace 5-11
istring 6-4
isupper 5-11
isxdigit 5-11

K

kflag 6-5

L

labels 4-11
lang 6-6
ldexp 5-25
left bracket ([) 5-48
library dictionary segment 1-13
library files 1-4, 1-13, 2-17, 2-18
  object files and 1-14
Library Index D-1–3
__LINE__ 4-6
LINK 2-5, 2-7, 2-10, 2-14–15, 3-2, 3-6
LinkEd 1-9, 1-10, 2-10, 2-17, 6-5
LINKED 1-16
linker 1-2, 1-3, 2-17

LinkName 2-14
lmalloc 5-34
load files 1-4, 1-6
  compact 3-7
  creating 3-6
  relocatable 1-6–7
load segments 1-4
  assigning 1-10
  object segments and 1-11
local symbol 1-13
log 5-16
log10 5-16
LOGIN 2-4
long 5-60
longjmp 5-53
lops 6-5
lseek 5-33

M

MacGen 1-15
MAIN 2-5
main 4-23
MakeLib 1-13, 2-17
malloc 5-34–35
memccpy 5-36
memchr 5-36
memcmp 5-36
memory images 1-7
Memory Manager 1-5, 1-7, 1-12, 1-16, 6-9
memory segment table 1-16
memset 5-36
merr 6-4
merrf 6-4
metasymbols 4-10
mflags 6-5, 6-6
minus sign (-) 5-7
modf 5-25
Monitor xii
MPW C
  APW C compared C-1–3
  Pascal-compatible function declarations C-2

N

NaNs 4-6, 4-7, 4-8, 5-6, 5-16, 5-21, 5-48, 5-61
native mode 1-4
nbyte 5-64
ndigit 5-14
nelem 5-35, 5-44
new desk accessories 3-8–10
NewHandle 5-34
nextfile 6-9
NEXT_WILDCARD 6-2, 6-8, 6-9
nitems 5-24
null character (\0) 5-29, 5-39, 5-48, 5-59
numeric constants, APW C 4-3
numeric environment 4-8

O

object code, compiling and assembling 3-5
object files 1-3, 1-4, 1-6
  library files and 1-14
  linking 3-6
object module format 1-4
object segments 1-8–9
  creating 1-9
  load segments and 1-11
ochar 6-12
offset 5-26, 5-33
oflag 5-38
onexit 5-37
opcodes 4-10
open 5-38, 5-55, 5-64
org 6-6

P

parms 6-4
partial compilation 1-9
pascal 4-12, 4-13, A-2
Pascal
  APW C and 4-12–18
  enumeration types 4-4
percent character (%) 5-39, 5-41, 5-47
period (.) 5-39
pflags 6-6
plus sign (+) 5-7
pointer 5-47, 5-50
pow 5-16
printf 4-8, 5-39–41, 5-55
Print Manager 3-9
ProDOS 1-17
ProDOS 8 xiii
ProDOS 16 xiii, 1-2, 2-11, 4-24, 6-4, 6-8, 6-9, 6-11
programs
  compiling and linking 2-7
  event-driven xi
  running 3-7
p2cstr 5-59
putc 5-41, 5-42, 5-55
putchar 5-42, 5-55
puts 5-43, 5-55
putw 5-42, 5-55

Q

qFlag 4-24
qsort 5-44
question mark (?) 6-9
QUIT 4-24

R

rand 5-45
READ_INDEXED 6-2, 6-7, 6-8
realloc 5-35
REDIRECT 6-2, 6-11
register 4-5
register variables, APW C 4-5
relocatable code segments 1-7
relocatable load files 1-6–7
relocation dictionaries 1-7
RENAME 6-9
reserved symbols 4-6
rewind 5-23, 5-26
right bracket (]) 5-49
rindex 5-59
RUN 2-7, 2-15–16

S

SANE xiii, 1-4
  APW C and 4-6–9
scanf 5-47–50, 5-55
scanset 5-48–49
segment 1-13, 4-23
segment body 1-11
segment header 1-11
segments 1-7. See also specific type
semicolon (;) 4-11, 6-10
setbuf 5-51–52, 5-56
setjmp 5-53
SET_LANG 6-2, 6-6
SET_LINFO 6-2, 6-3–6
SET_MARK 5-32
SET_VAR 6-2, 6-7, 6-9
setvbuf 5-51, 5-52, 5-55, 5-56
sfile 6-4
shell 1-2
SHELL-C 6-3
shell calls 6-1–12
shell commands 2-6–17
sin 5-61
sinh 5-54
65816 compiler xiii
C65C816, byte ordering 4-19
size 5-51, 5-52
sizeof 5-24
sizeof 5-44
skipChars 5-59
source 5-36
source code, writing and editing 3-3–4
source files 1-6
  editing 2-6
spanChars 5-59
sprintf 5-39, 5-41
sqrt(x) 5-16
srand 5-45
srcStr 5-58, 5-59
sscanf 5-47, 5-50
StandAlone 4-24

Standard Apple Numeric Environment. See SANE
Standard C Library 1-17, 5-1–64, 5-51, 5-59
  error numbers 5-3–4
Standard I/O Package 5-15, 5-37, 5-55, 5-57
standard linker 1-3
START-ROOT 4-23
static 4-10
static segments 1-13, 2-17
stderr 5-55, 5-56
stdin 5-27, 5-29, 5-47
stdio 5-55–57
StdIO-h 5-51, 5-55, 5-57
stdout 5-39, 5-55
STOP 6-2, 6-12
str 5-7
strcat 5-58
strchr 5-59
strcmp 5-58, 5-59
strcpy 5-58, 5-59
strcspn 5-59
stream 5-18, 5-23, 5-24, 5-39, 5-42, 5-43, 5-47, 5-62
strlen 5-59
strncat 5-58
strncmp 5-58, 5-59
strncpy 5-58, 5-59
strpbrk 5-59
strrchr 5-59
strspn 5-59
strtok 5-59
strtol 5-60
switch C-3
symbolic reference 1-6
System Loader 1-3, 1-5, 1-7, 1-12, 1-16

T

tan 5-61
tanh 5-54
tell 5-33
toascii 5-9
tokenStr 5-59
tolower 5-9
_toolErr 4-13, 5-4, 6-7
Tool-Locator C-2
toupper 5-9
trig 5-61
type 5-51

U

ungetc 5-62
unlink 5-63
unsigned 4-23
unsigned char 4-21
unsigned long 4-21
unsigned short 4-21

V

val 5-53
value 5-14, 6-7, 6-8
variable names, APW C 4-2–3
varname 5-28, 6-7, 6-8, 6-10
VERSION 6-2, 6-7
void 4-3, 4-19

W, X, Y, Z

WD65816 4-6
whence 5-33
white-space characters 5-47, 5-48
wildcard characters 6-8, 6-9
write 5-64
WRITE_CONSOLE 6-2, 6-12

YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS BEFORE USING THIS SOFTWARE. ANY DOWNLOADING, REPRODUCTION, COPYING OR OTHER USE OF THE SOFTWARE WILL CONSTITUTE ACCEPTANCE OF THESE TERMS AND CONDITIONS.

SINGLE-COMPUTER END USER SOFTWARE LICENSE AGREEMENT

APPLE COMPUTER, INC. ("Apple") provides this software and licenses its use. You assume responsibility for the selection of the software to achieve your intended results, and for the installation and use of, and results obtained from, the software.

LICENSE

Pursuant to this license you may:

1. Use the software only on a single Apple computer. You must obtain a supplementary license from Apple before using the software in connection with systems and multiple central processing units, computer networks or emulations on mainframe or minicomputers.
2. Download the software only on media that is compatible with Apple manufactured computers.
3. Copy the software into any machine readable form for backup purposes in support of your use of the software on the single Apple computer.
4. Transfer the software and license to another party with a copy of this Agreement provided the other party reads and agrees to accept the terms and conditions of this Agreement. If you transfer the software, you must at the same time either transfer all copies, whether in printed or machine-readable form, to the same party or destroy any copies not transferred. Apple grants a license to such other party under this Agreement and the other party will accept such license by its initial use of the software. If you transfer possession of any copy of the software, in whole or in part, to another party, your license is automatically terminated.

This software is protected by United States copyright law. You must reproduce the Apple copyright notice on any copy of the software.

THIS SOFTWARE MAY BE ELECTRONICALLY DISTRIBUTED ONLY BY AUTHORIZED ELECTRONIC DISTRIBUTORS. IT MAY BE DOWNLOADED ONLY FOR PERSONAL OR NON-COMMERCIAL USES ON APPLE COMPUTERS AND MAY NOT BE REDISTRIBUTED OR USED FOR COMMERCIAL PURPOSES WITHOUT AN EXPRESS SOFTWARE DISTRIBUTION LICENSE FROM APPLE. These licenses are available from Apple's Software Licensing Department.

YOU MAY NOT MODIFY, REVERSE COMPILE, DISASSEMBLE, NETWORK, RENT, LEASE, LOAN OR DISTRIBUTE THE SOFTWARE, OR ANY COPY, IN WHOLE OR IN PART. YOU UNDERSTAND THAT UNAUTHORIZED REPRODUCTION OF COPIES OF THE SOFTWARE OR UNAUTHORIZED TRANSFER OF ANY COPY OF THE SOFTWARE MAY SUBJECT YOU TO A LAWSUIT FOR DAMAGES, INJUNCTIVE RELIEF, AND ATTORNEY'S FEES.

Apple reserves all rights not expressly granted to you.

Export law assurances

You agree and certify that neither the software and documentation nor any direct product thereof (1) is intended to be used for nuclear proliferation or any other purpose prohibited by the United States Export Administration Act of 1979, as amended (the "Act") and the regulations promulgated thereunder, and (2) is being or will be downloaded, shipped, transferred or reexported, directly or indirectly, into any country prohibited by the Act and the regulations promulgated thereunder.

Government End Users

If you are acquiring the software on behalf of any unit or agency of the United States government, you agree that: (a) the software is "Commercial Computer Software" as that term is defined in Paragraph 27.401 of the DoD Supplement to the Federal Acquisition Regulations (the "Supplement") or is within the equivalent classification of any other federal agencies' regulations; (b) the software was developed at private expense, and no part of it was developed with goverment funds; (c) the government's use of the software is subject to "Restricted Rights" as that term is defined in clause 52.227-7013 (b)(3)(ii) of the Supplement or in the equivalent clause of any other federal agencies' regulations; (d) the software is a "trade secret" of Apple for all purposes of the Freedom of Information Act; and (e) each copy of the software will contain the following Restricted Rights Legend:

"Restricted Rights Legend"

Use, duplication or disclosure is subject to restrictions as set forth in subdivision (b)(3)(ii) of the Rights in Technical Data and Computer Software clause at FAR 52.227-7013. Manufacturer: Apple Computer, Inc. 20525 Mariani Avenue, Cupertino, California 95014.

You agree to indemnify Apple for any liability, loss, costs and expense (including court costs and reasonable attorneys' fees) arising out of any breach of the provisions of this Agreement relating to use by the government.

Term

The license is effective until terminated. You may terminate it at any time by destroying the software together with all copies. The license will also terminate upon conditions set forth elsewhere in this Agreement or if you fail to comply with any of the terms or conditions of this Agreement. You agree upon such termination to destroy all copies of the software.

Disclaimer of Warranty

The software is provided "as is" without warranty of any kind, either express or implied, with respect to its merchantability or its fitness for any particular purpose. The entire risk as to the quality and performance of the software is with you. Should the software prove defective, you (and not Apple or an Apple authorized representative) assume the entire cost of all necessary servicing, repair or correction.

Apple does not warrant that the functions contained in the software will meet your requirements or that the operation of the software will be uninterrupted or error free or that defects in the software will be corrected.

Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you. This warranty gives you specific legal rights and you may also have other rights which vary from state to state.

Limitation of Remedies

In no event will Apple be liable to you for any lost profits, lost savings or other incidental, special or consequential damages arising out of the use of or inability to use any software even if Apple or an authorized Apple representative has been advised of the possibility of such damages, or for any claim by any other party.

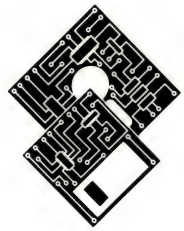
Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to you.

Apple's liability to you for actual damages for any cause whatsoever, and regardless of the form of the action, will be limited to the greater of $500 or the money paid for the software that caused the damages or that is the subject matter of, or is directly related to, the cause of action.

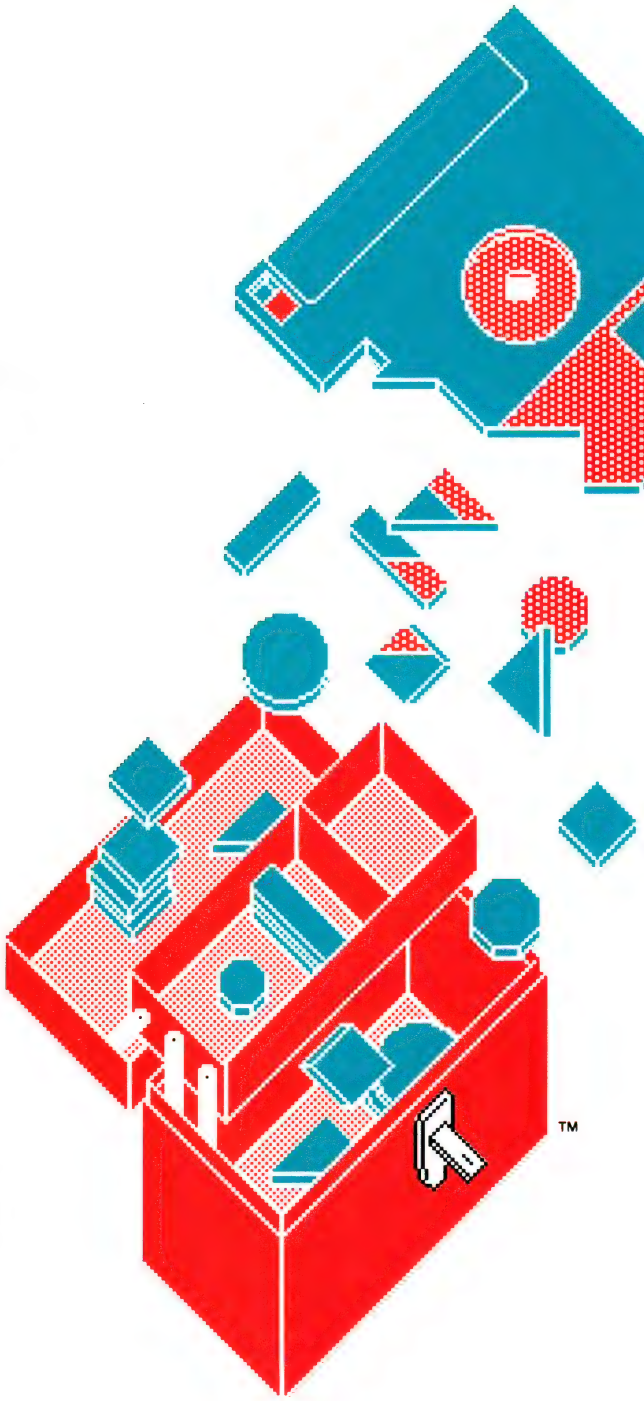
General

This Agreement, if any attempt to network, rent, lease, or sublicense the software, or, except as expressly provided in this Agreement, to transfer any of the rights, duties or obligations under this Agreement, becomes void.

The Agreement will be construed under the laws of the state of California, except for that body of laws dealing with conflict of laws. If any provision of this Agreement shall be held by a court of competent jurisdiction to be contrary to law, that provision will be enforced to the maximum extent permissible, and the remaining provisions of this Agreement shall remain in full force and effect.



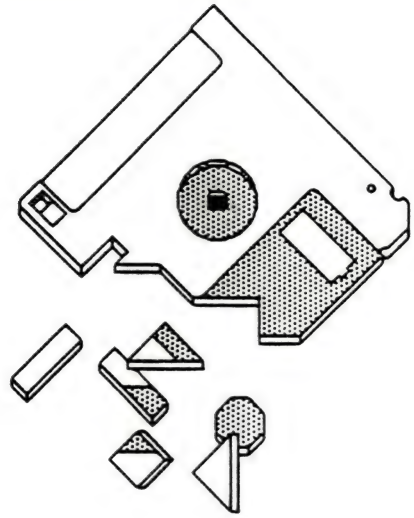
APPLE
PROGRAMMER'S
AND DEVELOPER'S
ASSOCIATION



APW C Interfaces Update

Version 1.0.1

APDA# K2S012



Guide to APW C Interfaces Update Release v.1.0.1

This APW C interfaces update consists of two items:

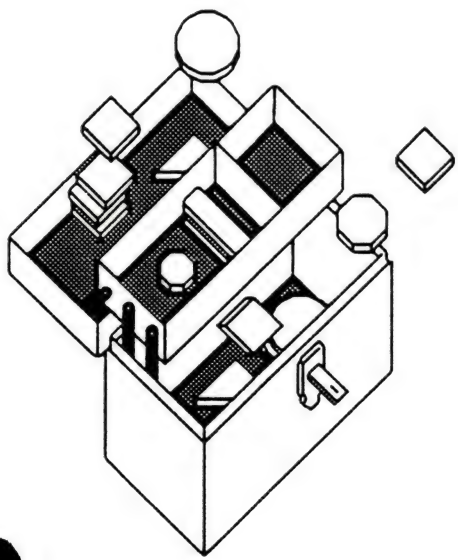
- A disk, /CLIBS, containing new APW C v.1.0.1 libraries and includes
- These release notes.

These release notes describe the changes from APW C v.1.0 to APW C v.1.0.1.

Note:

We believe we have done a thorough and complete job of testing of the product that you receive. However, please report any problems that you find with the APW software or documentation; you may use the enclosed "APW BUG REPORT" form. We appreciate your comments, even if you do not find any problems. Feel free to use the enclosed "APW SUGGESTIONS and EVALUATION" form to send us any recommendations for future APW software enhancements.

APW Product Manager
Apple Computer, Inc.
10500 N. De Anza Blvd., MS 27-S
Cupertino, CA 95014



APW C v.1.0.1

Only the Apple IIGS Toolbox Includes (the .H files in the CINCLUDE subdirectory) and the Standard C Library (CLIB), all located in the /APWC/LIBRARIES/ subdirectory, have been changed from APW C v.1.0 to APW C v.1.0.1. These changes have been made to update the toolbox includes to match the toolbox on Apple IIGS System Disk v.3.2, a programmer's kit for which is available separately from APDA (it consists of the system disk plus programming notes).

In addition to updates to existing files, the following files have been added to the CINCLUDE subdirectory:

ACE.H
MIDI.H

You should update Appendix B, p. B-1, of the *Apple IIGS Programmer's Workshop C Reference* to show these additions.

To update your working copy of APW C: Using the Finder, open the LIBRARIES subdirectory on your /CLIBS disk and copy the contents into the LIBRARIES subdirectory on your working disk.

APW SUGGESTIONS AND EVALUATION

After you have had a chance to become familiar with the APW software you are using, please complete this brief form.

What is your company name?_____

How long did it take for you to feel relatively comfortable with the APW environment?

_____

_____

What do you like best about APW? _____

_____

_____

_____

What do you like least about APW ?_____

_____

_____

_____

What would you like to see added to future versions of APW?_____

_____

_____

_____

General comments:_____

_____

_____

_____

_____

Please return this completed form to:

APW Product Manager, MS 27-S
Apple Computer, Inc.
10500 N. DeAnza Blvd.
Cupertino, CA 95014

APW BUG REPORT

Date_____
APW Version_____**System Disk Version**_____

Area (circle as many as required):

Shell
Compiler: C Pascal
Assembler
Library
APW Utility_____
Performance
Documentation

Configuration:

Hard Disk Size ____MB. Brand_____
Memory Size_____
Floppy Disks: _____3.5" _____5.25"

Bug Description:

_____

_____

_____

_____

_____

_____

_____

_____

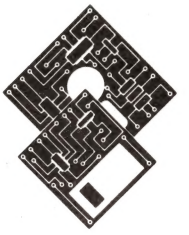
_____

Contact Information:

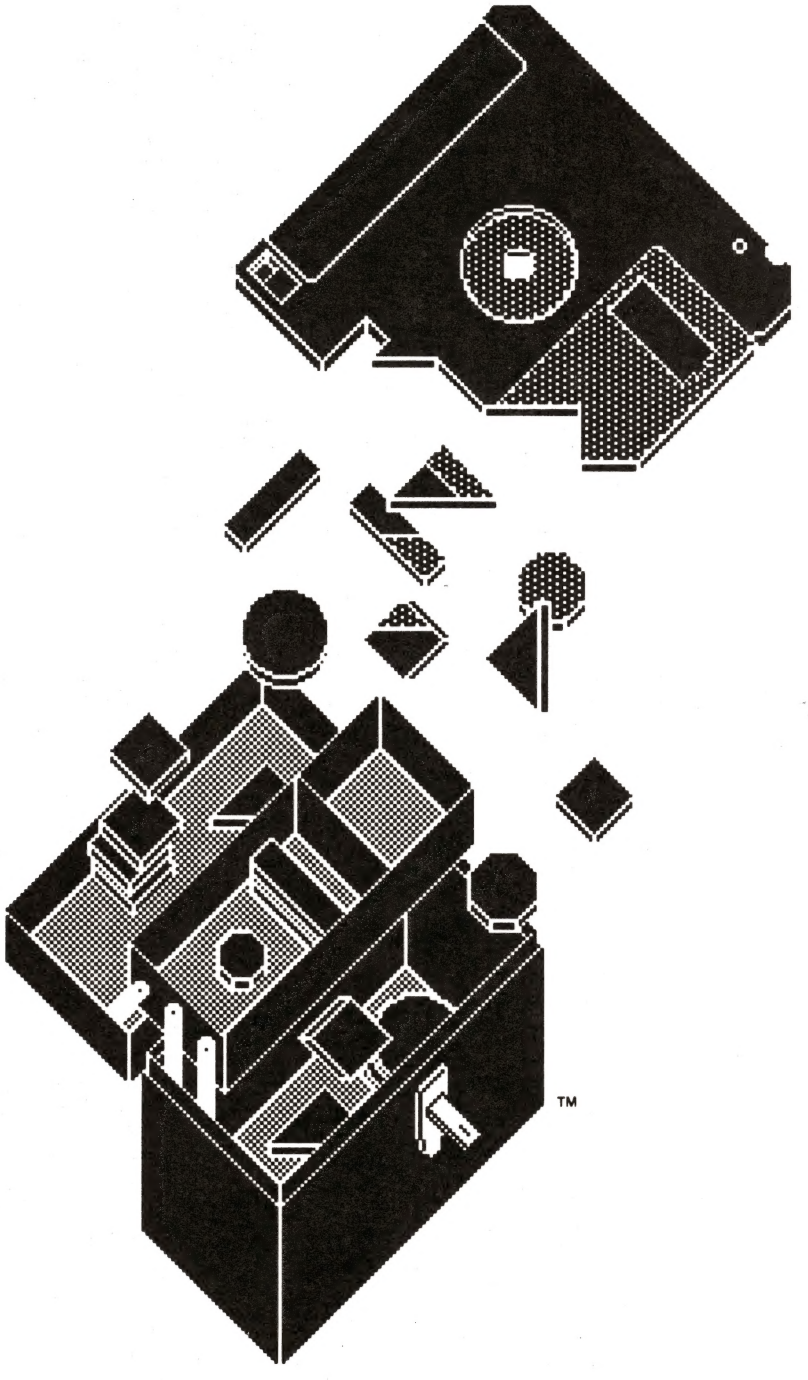
NAME_____ **Phone#**_____
Company Name_____
Address_____
City, State Zip_____

Please return this completed form to:

APW, Testing, MS 27-AQ
Apple Computer, Inc.
10500 N. DeAnza Blvd.
Cupertino, CA 95014



APPLE
PROGRAMMER'S
AND DEVELOPER'S
ASSOCIATION



™